A study of nested-relational joins in mediator-based distributed environments

Boersma, M.J.

Award date:
2011

Technische Universiteit Eindhoven University of Technology
Department of Mathematics & Computer Science
Database & Hypermedia Group

A study of nested-relational joins in mediator-based distributed environments

MSc Thesis

PUBLIC VERSION

Author	M.J. (Melle) Boersma	
Daily supervisors	dr. G.H.L. Fletcher	(TU/e)
	G. Bosma	(Triple R IT BV)
Supervisors	dr.ir. H.M.W. Verbeek	(TU/e)
	prof.dr. P.M.E. De Bra	(TU/e)

Eindhoven, July 2011

Table of Contents

List of Figures

List of Tables

List of Algorithms

Abstract

This thesis reports on the results of a study of the nested join operator in a mediator-based setting. As the rate at which data is accumulated increases, the need for integration of different (physical) Database Management Systems (DBMS) becomes apparent. The most fundamental and arguably important operation that is used in order to achieve this is the join-operator.

The nested-relational model is a flexible data model and can be used to describe data on practically all de facto data models such as relational, object-relational and XML-based DBMS.

Although joins are not new, very little work has been done in the area of the nested-relational join operator. Some authors [15, 18, 19] have proposed nested relational join operators, however an implementation has not been reported.

Information systems companies in their IT-landscape are often of a different make. This leads to difficulties concerning data integration. To solve these difficulties in an efficient way a mediation service, also known as a mediator, is introduced to their IT-landscape. The mediator facilitates communication between different databases and the end-user.

The focus of this study is the *design*, *implemention* and *evaluation* of a nested-relational join operator in a *mediator-based context*.

This study presents a generalized version of the join introduced by Garani & Johnson [15]. This generalized join is implemented in a mediator-based context, which poses some additional constraints on the implementation. Subsequently the implementation is evaluated and the results are presented.

Preface

This thesis is the final product of my master, Business Information Systems. It also marks the end of my life as a student. After having finished a bachelor's degree in Industrial Enigineering and Management Sciences, I chose for the master Business Information Systems (BIS). This was exactly what I wanted. Whereas in the bachelor, the Computer Science part was just 'one of those' areas which was touched upon, in the master it became a major part.

With the courses 'Information Retrieval' and 'Advanced Database Systems', Mykola Pechenizkiy and Toon Calders triggered my interest in the Databases and Hypermedia Group early on. And this interest was not unduly, this field is becoming more and more important as we, mankind, accumulate more and more data over time.

First of all, I would like to thank George Fletcher for his expertise and advice. I believe that I could not have been introduced in a better way to the academic field of Database Technology. I want to thank my company supervisor Gerco Bosma for his advice and feedback during the project and for showing me the clash between the academic world (theory) and the business side (practice). I would also like to thank Paul de Bra, Eric Verbeek and Lydia Zabel for their input in the process.

All of this would not have been possible without the love and support of my parents, Marja and Kerst, my sister, Maren and my girlfriend, Lonneke: Thank you for being there for me.

Chapter 1

Introduction

Around 500 BC the Greek philosopher Heraclitus said that 'change is the only constant'. The present day is no different: IT is ever-advancing. While these advancements present possible threats, they present even more possibilities and opportunities. From this constant flux however, one certainty can be derived: the amount of data we accumulate is ever growing.

In larger companies multiple systems fulfil business requirements of multiple divisions and/or business-units. Unfortunately these systems each have their standards for data storage and retrieval. In order to keep data up-to-date in multiple systems, the field of data integration flourishes since the 1980's, since keeping these systems synchronised is often essential for business operations. However, because of different storage formats and communication protocols, this is by no means an easy task. As a result, some organisations have become largely dependent on processes involving data mining and data integration where data originates from different sources.

The problem which is at the heart of this thesis is in line with the problem description in the previous paragraph. It is based on a real-life case of the company, introduced in Section 1.1, which also supervises this thesis. A more detailed version of the problem is given in Section 1.2. In Section 1.3 the research statement of this study is presented after which the contributions of this study are given in Section 1.4. The chapter concludes with an overview of the structure of the remainder of this thesis in Section 1.5.

1.1 Relevance to industry

Triple R IT BV[1] is a high-tech start-up founded in 2005 based at the High-Tech Campus in Eindhoven, The Netherlands. With their product – the DataSource

[1] See http://www.triplerit.com

Integrator – they provide data integration and aggregation solutions for multinationals. These companies often have an abundance of data in separate systems. As the amount of data grows, it is harder for companies to keep this data synchronised and to extract reliable, relevant and meaningful information from this data (e.g. KPIs).

As an example consider a production company that consists of three different departments: finance, production and sales. The finance department is responsible for invoicing and the financial statements of the company for which they use a software solution called Exact. The production department manages the state of their accepted orders, their due dates and resulting work force planning. This department uses a tailored SAP implementation, which is an Enterprise Resource Planning (ERP) system. The sales department is responsible for the order sales and the customer-relationship management of the company for which they use a software solution of ACT.

When these three departments do not share information and operate as independent silos any of the following problems could occur due to either the lack of information, or the use of outdated information:

- Billing a client too early or too late.
- Billing a client at a wrong address.
- Creation of an incomplete or wrong financial statement.
- Promising a client a too optimistic due date.

By integrating the data from these 3 departments the problems described above could be prevented. However, since these data sources use different means to access their data and use different formats for data storage, the integration of data is not a trivial operation.

Triple R IT facilitates the data integration need of companies with their product. In order to do so in the best possible manner, all software and hardware used are designed and custom-built in-house.

1.2 Context

This section was modified since it contained classified information.

A running example which is referred to in the remainder of this study is depicted in Tables 1.1, 1.2, 1.3 and 1.4. This data describes the aforementioned structure where a supplier (Table 1.1) has one or more locations (Table 1.2) which in turn have a set of products (Table 1.3) of which some have been revised over time (Table 1.4). The ID attributes (such as sID and $prID$) denote unique identifiers of tuples, or references to them. The *product* relation for instance has an $slID$ attribute. This attribute refers to a tuple within the *supplierLocation* relation to which it belongs.

Table 1.1: *The supplier relation.*

sID	supplierName
s1	supplier1
s2	supplier2

Table 1.2: *The supplierLocation relation.*

slID	sID	address	phone
sl1	s1	address1	235-7111317
sl2	s1	address2	192-3293137
sl3	s2	address3	414-3475359

Table 1.3: *The product relation.*

pID	slID	code	name	description	price
p1	sl1	code1	product1	description1	1.12
p2	sl1	code2	product2	description2	3.58
p3	sl2	code3	product3	description3	1.32
p4	sl2	code4	product4	description4	1.34
p5	sl3	code5	product5	description5	5.58
p6	sl3	code6	product6	description6	9.14
p7	sl3	code7	product7	description7	4.23

Table 1.4: *The articleRevision relation.*

prID	pID	oldCode	oldPrice
pr1	p1	codeX	3.37
pr2	p1	code1	6.10
pr3	p5	code5	9.87
pr4	p5	code5	2.58

In this section an introduction to the guiding case was given and a running example was introduced. In the next section the resulting research statement is presented.

1.3 Research question

The nested-relational model [17], a model used to store and represent hierarchical data, is a flexible data model and can be used to describe data on practically all de facto data models such as relational, object-relational and XML-based database management systems (DBMS).

As the amount of data grows, manual lookups of specific data elements and their manipulation can be quite laborious and error prone. For this reason DBMS were developed. These systems allow to retrieve and transform specific sets of information based on user requests. Arguably the most important operation within the field concerns combining different types of information. This operation is executed by a so called 'join operator'.

Although joins are not new, relatively little effort has been made in the area of the empirical study of the nested-relational join operator. Some papers [15, 18, 19] presented nested relational join operators, however no implementation has been reported.

Next to that, the nested join operation has not been implemented in a distributed setting in which there is a central service which facilitates the execution of this operation. This central service is also known as a mediator. The mediator and its context are interesting because with the ever increasing amount of systems available on the market today the number of interoperability requirements increases. Next to that, the amount of data processed by these systems increases. One solution to deal with both interoperability requirements and lack of processing power is to introduce a mediation service.

The previous observations have resulted in the following research question:

> How can the nested relational join operator be implemented in a mediator based context?

This leads to several sub research questions:

1. What is the result of a join operation on nested-relational data?
2. How can a nested-relational join operator be implemented?
3. What are the properties that affect the performance of the nested relational join operator?

The following steps were taken in this study to answer the aforementioned research question and sub-questions:

1. Design of an algorithm for non-distributed nested-relational join based on earlier work [15, 18, 19].
2. Implementation and validation of the non-distributed algorithm.
3. Design of an algorithm for the distributed nested-relational join.
4. Implementation and validation of the distributed design.
5. Set-up of an experiment for empirical analysis of the designed distributed algorithm.
6. Execution of the experiment.
7. Analysis of the results.

This section introduced the research question and sub-questions. A number steps were undertaken, to answer these questions. This lead to several contributions of this study which will be presented in the next section.

1.4 Contributions

This section was modified since it contained classified information.

This section discusses the contributions of this study. Since this study is based on an actual use-case within the supervising company, this study will contribute to the academic field and add business value on a practical level.

Garani & Johnson [15] identify six cases of the nested-relational join. Their proposal is the first extension of the nested-relational join operator which is not limited to joining nested structures in any way.

First of all a generalisation of these six cases of the join is presented. This single generalisation reduces the complexity of implementation and validation for the different cases of a nested join operator.

Second, a first implementation of a local Garani-Join algorithm is presented extended to a mediator-based context, which imposes several design restrictions.

Third, the implementation is evaluated. This is the first reported empirical evaluation of such a nested-relational join operator. Only by implementation, bottlenecks can be identified and optimizations can be made. This thesis is a first step towards such optimizations.

1.5 Thesis outline

This section gives a brief overview of the content of the other chapters of the thesis.

Chapter 2 presents background information related to the subject and the research statement. In this chapter theoretical concepts are introduced which will be used throughout the rest of the chapters.

Chapter 3 discusses the nested-join operator in more detail. Since this is the main focus of this thesis, a more in-depth analysis is in order. This chapter includes a generalisation of the nested-join operator. Finally, several implementations of join algorithms will be presented.

Chapter 4 describes the experimental set-up. The input parameters, output parameters, data-sets and experiment implementation are presented here as well.

Chapter 5 will contain the evaluation of the experiment. After running the experiment, several observations have been made. These observations can be found in this chapter.

Chapter 6 will contain the conclusions of this study, an overview of the contributions and a listing of future work.

In this section the outline of the thesis is presented. In the next chapter, background information related to the academic field is presented.

Chapter 2

Background

In this chapter an overview of literature which is the basis of the aforementioned research statement is presented. First the foundation of database technology, relational algebra, is discussed alongside the running example presented in Chapter 1. The notation introduced here will be used throughout the rest of the thesis.

Since the research focusses on the join operator, this subject will be discussed in more detail and several extensions will described. After that, leaving the declarative field behind, we focus on the implementation of query processing in general and the join operator in specific. This subject is extended to a distributed environment, i.e. a mediator-based context and several considerations and trade-offs are discussed.

2.1 Relational algebra

Around the 70's the first ERP systems were developed and data began to accumulate. In order to reason in a formal manner about this data, Codd introduced both the relational model, which is a set of criteria also known as First Normal Form (1NF or NF), and *relational algebra* [10]. Based on set theory, relational algebra can be used as a means to reason and describe data structured according to the relational model.

Next several concepts of relational algebra are introduced. Just as in set theory, a **tuple** is an ordered list of values, such as *(B1, Mathematics, Peter)*. This example describes a book *B1* on the subject *Mathematics* owned by the person *Peter*. The nouns used to describe the values of the tuples are known as **attributes**, in this case *{book, subject, person}*. A set of tuples with the same attributes is called a **relation**. A **scheme** is a tree representation of the relation with its attributes. An example of a scheme is depicted in Figure 2.1. In this case an extended tree structure of nested relations of a supplier as presented in Section 1.2 is depicted.

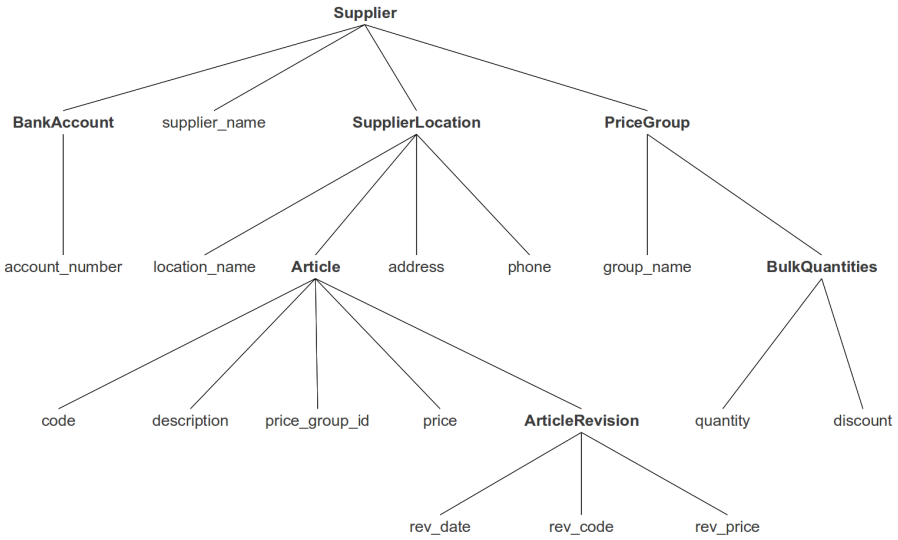


Figure 2.1: *An example of a scheme*

In order to describe operations on relations and attributes in a formal manner, a set of relational operators was introduced [10]. In Table 2.1 the basic operators of relational algebra are described [10, 22, 24]. The nest and unnest operators are discussed in Section 2.2.

The *select*, *project* and *rename* operators are called *unary* operators since they require a single operand. In the same fashion the other operators are *binary* operators since these operate on two operands.

In order to reduce the length of query statements, the binary **Natural-Join operator** (also known as the Join operator or simply 'Join') was introduced. It is defined as

$$R \bowtie S \equiv \Pi_{(attr(R) \cup attr(S))}(\sigma_P(R \times S))$$

where $attr(X)$ denotes the set of attributes of relation X and predicate P is a set of equations, defined as

$$P \equiv \{R[x] = S[x] | x \in (attr(R) \cap attr(S))\}.$$

Informally, the join operator matches two sets of tuples on a number of shared attributes.

A graphical representation of the most common relational operators is depicted in Figure 2.2. In this figure the rectangles denote the relations. Within those, the marked areas are affected by the described operation. The arrows denote the transition from input to output. R and S denote the relations participating in the operation.

Other operators were defined to aid in the formulation of queries. For an overview of these additional relational algebra operators see [24, p.58].

Table 2.1: *Relational algebra operators*

Operation	Notation	Description
Select	$\sigma_P(R)$	The select operator returns the subset of tuples in R for which the predicate P holds.
Project	$\Pi_A(R)$	This operator returns a relation including only the attributes in attribute list A.
Rename	$\rho_x(R)$	Renames relation R to x. In this case, x can be a relation name or a relation name combined with attribute names $x(A_1, ..., A_n)$ where A_i is an attribute name.
Union	$R \cup S$	The union operator appends the tuples of S to R and returns the result. Note that relations R and S should have the same attributes.
Intersection	$R \cap S$	The intersection operator returns the set of tuples which exist in both relations R and S.
Difference	$R - S$	The difference operator behaves like set difference, it removes the tuples that exist in S from R.
Cartesian product	$R \times S$	Combines every tuple from R with every tuple from S by adding the attributes from S to those from R.
Division	$R \div S$	Let $R \supseteq S$. The division operator returns a unique set of tuples with attributes that do not exist in S for which the attributes in common are in the complete set of S.
Nest	$\nu_A(R)$	Nests relation R on attribute(s) A. Results in a nested relation.
Unnest	$\mu_A(R)$	Unnests relation R on attribute(s) A. Inverse of the nest operation.

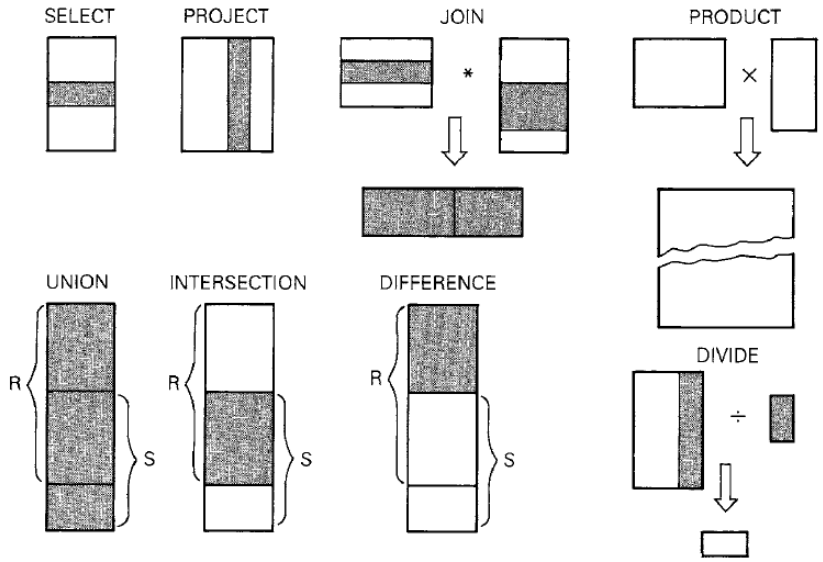


Figure 2.2: *An graphical representation of common relational operators from [10]*

2.2 Nested relational algebra

Up until now, the values of attributes are **atomic**, i.e. they only contain literal values. Jaeschke & Schek [17] proposed the nested relational model also known as the Non First Normal Form (NF^2) as an extension to the relational model. In this model, attributes are either atomic or **nested**, i.e. within tuples the values of these attributes are a relation by themselves. This extension allows describing hierarchical data.

Important extensions for dealing with nested relational data are the *nesting* and *unnesting* operations, described in Table 2.1. These operations can be used for transformations between NF^2 and NF. Nesting and unnesting operations are denoted by $\nu_A(R)$ and $\mu_A(R)$, respectively.

As an example, consider a projection of the pID and $slID$ attributes of the *product* relation from Table 1.3. This projection is depicted in Table 2.2. Performing a nest operation on the pID column ($\nu_{pID}(R)$ or $\nu_{\$1}(R)$) would result in the relation depicted in Table 2.3.

Table 2.2: *A projection of attributes pID and $slID$ of the product relation.*

pID	slID
p1	sl1
p2	sl1
p3	sl2
p4	sl2
p5	sl3
p6	sl3
p7	sl3

Table 2.3: *The result applying the nest operation on the pID attribute of the relation described in Table 2.2.*

pID	slID
{(p1), (p2)}	sl1
{(p3), (p4)}	sl2
{(p5), (p6), (p7)}	sl3

Although the nest and unnest operations are a welcome extension to the set of operators, their usage can cause a large explosion of data, especially the usage of the unnest operator.

The introduction of nested relational data required an extension of the natural join operator. Roth *et al.* [23] describe an extension of the set of basic operators to be used in a nested environment. They present the so-called *extended natural join* ($\bowtie^e$) which is a recursive application of the join operator on nested

attributes. The extended natural join of relations r and q with schemes R and Q and common attributes A_c is defined as:

$$
\begin{aligned}
r \bowtie^e q \equiv \{t | (\exists t_r \in r, \exists t_q \in q) \\
\wedge (t[attr(R) - A_c] = t_r[attr(R) - A_c]) \\
\wedge (t[attr(Q) - A_c] = t_q[attr(Q) - A_c]) \\
\wedge (t[A_c] = t_r[A_c] \cap^e t_q[A_c]) \\
\wedge (t[A_c] \neq \varnothing) \\
\}
\end{aligned}
\qquad \text{(Eq. 2.1)}
$$

In equation Eq. 2.1 the extended set intersection ($\cap^e$) is defined informally as the complete equality of all (nested) attributes. A complete definition is described by Roth *et al.* [23]. This extension however, is limited to joining relations at the top level of the scheme tree.

This limitation was addressed by Liu & Chirathamjaree [18] who present a new operator called the P-join. However the operator described by the authors is limited to joining what they call *selection-comparable nodes*. In order for two nodes to be selection comparable one of the two nodes has to be a child of an ancestor of the other node. Unfortunately their proposed algorithm is flawed in case a join on multiple attributes distributed over multiple levels is performed. Although the authors claim their operator works correct, the resulting nested attributes point to specific points in a global scheme, thereby causing the scheme-tree to be transformed to a lattice.

Fortunately, Garani & Johnson [15] provided a structural analysis of all cases which occur when joining nested relational data. Since this is the basis for this thesis, their research will be discussed in the next chapter.

2.3 Implementation of the natural join

Relational algebra can be used to declare operations on one or more relations. In order to obtain these results, relational operators will need to be translated to instructions at a lower level. In this section the two most common implementations of the join operation will be presented: the nested-loop join and the sort-merge join.

In the described algorithms the equality and inequality statements operate on the set of common attributes. As an example, the *matches* operation in Algorithm 2.1 tests for equality of two tuples. In this algorithm, $attr(t)$ denotes the set of attributes of tuple t and $t[a]$ refers to the set of values of tuple t for a set attributes a.

The nested-loop join is the most straightforward implementation of the natural join operator. It iterates over two nested loops and compares each tuple of relation R to each tuple in relation S as shown in Algorithm 2.2. The comparison is based on the *matches* operations as described in Algorithm 2.1. In this case

Algorithm 2.1 The *matches* operation for tuples t_1 and t_2 adapted from [24]

```
1: for all a ∈ (attr(t1) ∩ attr(t2)) do
2:   if t1[a] ≠ t2[a] then
3:     return False
4:   end if
5: end for
6: return True
```

the *block* version of the nested loop join is presented. This implementation is based on the notion that the memory capacity of the system performing the algorithm is likely to be insufficient to be able to contain the two complete relations during the operation. Therefore, each relation is composed of a set of blocks and accessed on block-per-block basis. The assumption in this algorithm is that the two blocks (B_R and B_S) and the temporary result set (T) fit into memory. The complexity of this algorithm is $O(N^2)$, where N is the maximum number of tuples of R and S.

Algorithm 2.2 The block nested-loop join over relations R and S.

```
1: T := {}
2: for all B_R ∈ R do
3:   for all B_S ∈ S do
4:     for all t_r ∈ B_R do
5:       for all t_s ∈ B_S do
6:         if t_r = t_s then
7:           T := T ∪ {(t1 ∪ t2)}
8:         end if
9:       end for
10:    end for
11:  end for
12: end for
```

The sort-merge join, proposed by Blasgen & Eswaran [4] strives to reduce the space and time complexity of the nested-loop join. By sorting both relations before performing the join, a large number irrelevant match operations can be avoided thereby reducing complexity to $O(NlogN)$.

The sort-merge join algorithm works as follows. First the both relations are sorted based on their common attribute(s). Then all (consecutive) tuples on the left hand which match are grouped together, up to the first tuple which has different values for the common attributes. After that the right hand side relation is scanned. This scan continues as long as the common attributes on the right hand side are ranked lower than those on the left hand side. As soon as this is no longer the case, the tuple on the right hand side is tested for equality with the set of tuples on the left hand side. If they are equal, the tuple is added to the result set and the next tuple of the right hand side is checked for equality. When this tuple is not equal, its rank is higher than any of the tuples in the set on the left hand side. Another pass is then made with a new set of tuples on the left hand side.

The blocked version complicates this algorithm considerably. If a subsequent block is requested on either side, and this block has the same common attribute values as the last tuple in the previous block, some parts of the other relation need to be scanned again. Therefore the last block and offset of any matching operation needs to be stored, so this position can be restored when a matching operation is interrupted by block boundaries.

Algorithm 2.3 depicts the sort-merge-join. In this algorithm r_i denotes the i^{th} element of ordered relation R and $|R|$ the cardinality of relation R, i.e. the number of blocks contained in relation R. Furthermore, $sort(R, A)$ is defined as a sort operation on attributes A of relation R. Again a *block* version of the algorithm is presented for same reasons outlined earlier. Let B_{Ri} denote the i^{th} block of relation R and let $|B_{Ri}|$ denote the cardinality of block B_{Ri}. The algorithm is roughly based on the 'unblocked' algorihm outlined by Sliberschatz *et al.* [24].

In this section, two blocked implementations of join algorithms were reviewed. However, these algorithms are meant to be used in a local setting; that is in a centralised location. As a step towards the distributed execution of join operations, distributed query processing is discussed in the next section.

Algorithm 2.3 The block sort-merge join for relations R and S.

```
1:  output := {}
2:  i := 0;   i_last := −1
3:  j := 0;   j_last := −1
4:  m := 0;   m_last := −1
5:  n := 0;   n_last := −1
6:  sort(R, attr(R) ∩ attr(S))
7:  sort(S, attr(R) ∩ attr(S))
8:  while (i < |B_Rm| ∨ m < |R|) ∧ (j < |B_Sn| ∨ n < |S|) do
9:    if i = |B_Rm| then
10:     m := m + 1
11:     i := 0
12:     if j_last > −1 then
13:       j := j_last;   n := n_last
14:     end if
15:   end if
16:   if j = |B_Sn| then
17:     t := s_j
18:     n := n + 1
19:     j := 0
20:     if s_j = t then
21:       i := i_last;   m := m_last;   j_last := −1
22:     end if
23:   end if
24:   i_begin := i
25:   P := {r_i}
26:   while (i + 1 < |B_Rm|) ∧ (r_i = r_{i+1}) do
27:     P := P ∪ {r_{i+1}}
28:     i := i + 1
29:   end while
30:   while (j < |B_Sn|) ∧ s_j < r_i do
31:     j := j + 1;   j_last := −1
32:   end while
33:   while (j < |B_Sn|) ∧ (r_i = s_j) do
34:     if j_last = −1 then
35:       j_last := j;   n_last := n
36:     end if
37:     if i_last = −1 then
38:       i_last := i_begin;   m_last := m
39:     end if
40:     for all p ∈ P do
41:       output := output ∪ {r_i ∪ s_j}
42:     end for
43:     j := j + 1
44:   end while
45:   if r_i < s_{j−1} then
46:     i_last := −1
47:   end if
48: end while
```

2.4 Distributed query processing

In the previous section, local implementations of the join operator were described. Often queries are performed on data from multiple sources distributed over several physical locations. In this section query processing in such a non-centralised environment is discussed by presenting two different approaches for performing the join: a naive approach and the semi-join.

In this section, consider two different physical locations, called sites S_1 and S_2. These sites contain relations R_1 and R_2 respectively. A join is performed between relations R_1 and R_2 where a set of common attributes $A_c := A_{R1} \cap A_{R2}$ are compared. Furthermore, assume $|R_1| < |R_2|$, i.e. the relation at S_1 is smaller than the relation at S_2.

The *naive* approach [24] is to execute the query at the larger relation, send the smallest relation (R_1) to the site of the larger one (S_2) and to perform the join at the site of the larger relation.

1. Request R_1 from S_1.
2. Ship relation R_1 from S_1 to S_2.
3. Perform a local join $result \leftarrow R_1 \bowtie R_2$.
4. Return *result*.

Note that this local join can be implemented using any of the previously discussed joins: the nested-loop join, the sort-merge join or any extended natural join for that matter, depending on the structure of the data.

The *semi-join* originally proposed by Bernstein & Chiu [3] aims at reducing the number of bytes transmitted by shipping a projection of the common attributes of R_1 to S_2. A join is performed solely on these attributes, after which an intermediate result is sent back to S_1. After receiving the intermediate result, it is again used in a join to obtain the final result. This process results in the following steps, based on Silberschatz *et al.* [24]:

1. At S_2 request $\pi_{A_c}(R_1)$ from S_1.
2. At S_1 compute $temp_1 \leftarrow \pi_{A_c}(R_1)$.
3. Ship $temp_1$ from S_1 to S_2.
4. At S_1 request $temp_1 \bowtie R_2$.
5. At S_2 Calculate $temp_2 \leftarrow temp_1 \bowtie R_2$.
6. Ship $temp_2$ from S_2 to S_1.
7. At S_1 compute $result \leftarrow R_1 \bowtie temp_2$.
8. Return *result*.

In the next section the distributed environment related to this research is presented. Within this environment the approaches discussed above are used.

2.5 Mediator-based systems

This section was modified since it contained classified information.

This section discusses the systems used for distributed query processing such as those described in the previous section. Initially mediator-based systems are presented, after which a broader overview of related literature is given. The term *mediator* is based on and used interchangeably a the definition of a *mediation service* [27] which is described as:

> "[A mediation service] covers value-added processing on query text and resulting content as query reformulation and distribution; and filtering, integrating and processing the content of the resulting data, generating new, denser or more relevant information."

Accordingly, a **mediator** is defined as a component that provides mediation services in response to a client. In Figure 2.3 a mediator is depicted in its typical context. In this picture *wrappers* are responsible for the translation and imple-

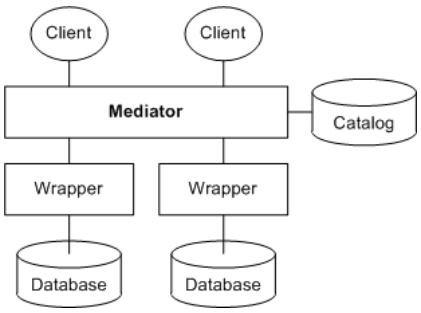


Figure 2.3: *Wrapper architecture based on [26]*

mentation of database specific functionality, also known as the *capabilities* of the data source. A *catalog* is stored which contains a global scheme of the separate database systems. A *client* connects to the database requesting information. The mediator translates their requests to appropriate queries and optionally performs some final operations before sending a response to the client.

Several systems for managing heterogeneous data sources (i.e. data sources with different capabilitites) have been proposed, amongst them are DISCO [25], MIND [13] and Garlic [7]. Some of these systems take the data source's *capabilities* into account. These refer to the available operations and their operands of a certain data source. For example, some data sources might not support aggregate operations such as `COUNT` or `SUM`. In this case a wrapper should support this functionality.

As a result, the internal language used by the mediator should be rich enough to support a wide variety of operations. For instance the Garlic, MIND and DISCO frameworks are based on ODMG's object data model [2] which is a specification for the communication and storage of hierarchical data.

Collet & Vu [12] present the Query Broker Framework (QBF). With the QBF the authors propose a generalised solution to simplify the extension of query functionality in a heterogeneous setting. They do so by defining a framework that encapsulates distributed query optimisation and monitoring. A monitor responds to simple event-condition-action or event-action rules. Furthermore, capability and query context of data sources is taken into account. In the QBF the context of a query is defined as a set of constraints relating to parameters such as system load or network traffic. The query itself is represented as a set of nodes forming a tree cf. the arity of the query operators described in Section 2.1. An example of such a tree is given in Figure 2.4.

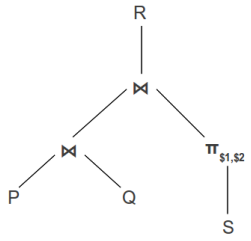


Figure 2.4: *An example of a query displayed as a tree*

In this chapter several concepts ranging over various levels within the field of data integration were presented. First a basic introduction to relational algebra and operators was given. At a lower extent general implementations of natural joins were presented for unnested relations. Subsequently, these joins were brought to a distributed environment. Finally, several systems for managing and executing both ad-hoc and planned queries were reviewed.

In the next chapter, an analysis of nested relational joins is presented.

Chapter 3

Analysis of the nested join

In the previous chapter a general introduction to the academic field was presented. In this chapter, several theoretical aspects will be discussed in more detail, as they provide the foundation for the thesis. First an analysis of the different joins as described by Garany & Johnson [15] is presented. Their work is a clear and concise summary of the research done in this field. Next, a single generalised version of these joins is derived. An implementation for this generalised version is presented subsequently in Section 3.2. This generalised version is implemented (see Chapter 4) and forms the basis for the evaluation in Chapter 5, something which has not been done before.

3.1 Analysis of the G-Join

Garani & Johnson [15] identify six cases of the nested relational join. They distinguish the cases by looking at the type of the attribute (atomic or nested) and the level at which the participating attributes are within the scheme tree such as the one presented in Section 1.2, Figure 2.1. These six cases are depicted in Table 3.1.

In order to distinguish between parent (R) and child (R_i) relations in the scheme tree, a *join-path* (L) is introduced. This *join-path* is defined as being either *1)* $L = \varnothing$ at the level of the participating relation or *2)* $L = R_i L_i$ otherwise . In the above definition, L_i denotes the remainder of the join-path at depth i.

In the upcoming descriptions, $\bowtie_N$ denotes the definition of the join operator according to case N as depicted in Table 3.1.

Case 1

Although relations might contain nested attributes, these are not in the set of common attributes and hence not participating in the join. Therefore

Table 3.1: *The six cases of the join operation.*

	1^{st} relation		2^{nd} relation		
Case	Attribute	Level	Attribute	Nesting	
1	atomic	top	atomic	top	
2	atomic	not top	atomic	top	
3a	atomic	not top	atomic	not top	(same level)
3b	atomic	not top	atomic	not top	(not same level)
4	nested	top	nested	top	
5	nested	not top	nested	top	
6a	nested	not top	nested	not top	(same level)
6b	nested	not top	nested	not top	(not same level)

relations are joined as if they were atomic, i.e. nested attributes appear in the join result unmodified.

In equation Eq. 3.1 a formal definition of the join is given as defined in [15]. Relations r and q contain attributes $\{R_1, R_2, \ldots, A_1, \ldots, A_j, \ldots, R_n\}$ and $\{Q_1, Q_2, \ldots, A_1, \ldots, A_j, \ldots, Q_m\}$ respectively. In these attribute sets, $\{A_1, \ldots, A_j\}$ denotes the set of common attributes.

$$
\begin{aligned}
r \bowtie_1 q \equiv \{t | (\exists t_r \in r, \exists t_q \in q) \quad & \text{(Eq. 3.1)} \\
& \wedge (t[attr(R) - \{A_1, \ldots, A_j\}] = t_r[attr(R) - \{A_1, \ldots, A_j\}]) \\
& \wedge (t[attr(Q) - \{A_1, \ldots, A_j\}] = t_q[attr(Q) - \{A_1, \ldots, A_j\}]) \\
& \wedge (t[A_1, \ldots, A_j] = t_r[A_1, \ldots, A_j] = t_q[A_1, \ldots, A_j]) \\
& \wedge (t[A_1, \ldots, A_j] \neq \varnothing) \\
& \}
\end{aligned}
$$

Case 2

Case 2 considers a join operation on atomic attributes where one of the attributes is not at the top level. This is a recursive application of the join operator until the join-path is empty. In this case the join as defined in *case 1* is executed.

Again, consider relations r and q with attributes $\{R_1, R_2, \ldots, R_i, \ldots, R_n\}$ and $\{Q_1, Q_2, \ldots, A_1, \ldots, A_j, \ldots, Q_m\}$ where $\{A_1, \ldots, A_j\}$ are the attributes in common, and these common attributes are on the end of the join-path of R_i, i.e. R_i is the ancestor of $\{A_1, \ldots, A_j\}$. Let rL denote a relation r with join path L. The definition of equation Eq. 3.2 is slightly adapted from the authors definition [15] since they did not include the difference between the first and the second case formally. Note that in the following definition, R_i is a nested attribute and that hence $t[R_i]$ refers to a set of

tuples.

$$
\begin{aligned}
rL \bowtie_2 qM \equiv\ & r(R_iL_i) \bowtie_2 q(Q_jM_j) \qquad \text{(Eq. 3.2)} \\
\equiv\ & \{t|(\exists t_r \in r) \\
& \wedge (t[attr(R) - \{R_i\}] = t_r[attr(R) - \{R_i\}]) \\
& \wedge ((L \neq \varnothing \wedge t[R_i] \neq \varnothing \wedge (t[R_i] = t_r[R_i]L_i \bowtie_2 q)) \\
& \quad \vee (L = \varnothing \wedge t[R_i] \neq \varnothing \wedge (t[R_i] = t_r[R_i] \bowtie_1 q))) \\
& \}
\end{aligned}
$$

Case 3a

This case defines the join between atomic attributes which are on equal levels, but not at the top level. Let r and q be nested relations with schemes $R = \{R_1, R_2, \ldots, R_i, \ldots, R_n\}$ and $Q = \{Q_1, Q_2, \ldots, Q_j, \ldots, Q_m\}$. Let R_i and Q_j be the ancestors of a common set of atomic attributes at a specific but equal depth on their respective join-paths. Furthermore, let L and M be the join-paths of r and q respectively. The join for this case is then defined as:

$$
\begin{aligned}
rL \bowtie_{3a} qM \equiv\ & r(R_iL_i) \bowtie_{3a} q(Q_jM_j) \qquad \text{(Eq. 3.3)} \\
\equiv\ & \{t|(\exists t_r \in r, \exists t_q \in q) \\
& \wedge (t[attr(R) - \{R_i\}] = t_r[attr(R) - \{R_i\}]) \\
& \wedge (t[attr(Q) - \{Q_j\}] = t_q[attr(Q) - \{Q_j\}]) \\
& \wedge ((L_i \neq \varnothing \wedge t[R_iQ_j] \neq \varnothing \wedge t[R_iQ_j] = (t_r[R_i]L_i \bowtie_{3a} t_q[Q_j]M_j)) \\
& \quad \vee (L_i = \varnothing \wedge t[R_iQ_j] \neq \varnothing \wedge t[R_iQ_j] = (t_r[R_i] \bowtie_1 t_q[Q_j]))) \\
& \}
\end{aligned}
$$

Case 3b

This is the most complex case on atomic attributes since a join is performed on two attributes which are neither at the top relation, nor at the same level.

In essence, the result of this join consists of all tuples found in the common attributes, their ancestors and all relevant tuples of attributes not participating in the join. The relation which has the common attributes at a lower level in its scheme tree is included in the other.

In the following definitions P_{yx} denotes attribute number x from the set of P's attributes which is at depth y in the overall scheme. Moreover, $P_{yx}(P_{(y-1)1}, \ldots, P_{(y-1)n})$ denotes that P_{yx} is a nested attribute with n child attributes (atomic or nested). The collection of attributes of P at depth y is denoted by P_y.

Consider nested relations r and q again defined by schemes R and Q where the set of common attributes $A_11, \ldots, A_1z$ is present at different levels in the two scheme trees. In essence it describes two nested schemes (R and Q) which contain a set of common nested attributes $(A_{11}, \ldots, A_{1z})$

somewhere within the structure.

$$
\begin{aligned}
\text{Let } R = \{&R_{i1}, \ldots, R_{iw}(\\
&R_{(i-1)1}, \ldots, R_{(i-1)x}(\\
&\ldots(\\
&R_{21}, \ldots, R_{2y}(\\
&R_{11}, \ldots, A_{11}, \ldots, A_{1z}, \ldots, R_{1k} \\
&), \ldots, R_{2l} \\
&), \ldots), \\
&\ldots, R_{(i-1)m}, \\
&), \ldots, R_{in}\}
\end{aligned}
$$

$$
\begin{aligned}
\text{and } Q = \{&Q_{i'1}, \ldots, Q_{i'w'}(\\
&Q_{(i'-1)1}, \ldots, Q_{(i'-1)x'}(\\
&\ldots(\\
&Q_{21}, \ldots, Q_{2y'}(\\
&Q_{11}, \ldots, A_{11}, \ldots, A_{1z}, \ldots, Q_{1k'} \\
&), \ldots, Q_{2l'} \\
&), \ldots), \\
&\ldots, Q_{(i-1)m'}, \\
&), \ldots, Q_{i'n'}\}
\end{aligned}
$$

In the above definition $i, i', w, w', x, x', y, y', l, l', m, m', n, n'$ are positive integers and not equal in general. Furthermore assume that $i < i'$ without loss of generality.

Note that this is a more general definition than the one defined by the authors [15, def. 11] since they decided to present two scheme trees with participating relations at the first attribute only. It is therefore also a bit more verbose. As in the previous cases, let L and M be the join-paths of relations r and q respectively.

The join for this case can be defined as:

$$
\begin{aligned}
rL \bowtie_{3b} qM &\equiv r(R_iL_i) \bowtie_{3b} q(Q_jM_j) \qquad \text{(Eq. 3.4)} \\
&\equiv \{t|(\exists t_r \in r, \exists t_q \in q) \\
&\quad \wedge (t[Attr(Q_{i'}(\ldots(Q_{i+1})))] = t_q[Attr(Q_{i'}(\ldots(Q_{i+1})))]) \\
&\quad \wedge (t[attr(Q_i) - \{Q_{iw'}\}] = t_q[attr(Q_i) - \{Q_{iw'}\}]) \\
&\quad \wedge (t[attr(R_i) - \{R_{iw}\}] = t_r[attr(R_i) - \{R_{iw}\}]) \\
&\quad \wedge (t[R_{iw}Q_{iw'}] = (t_r[R_{iw}]L_i \bowtie_{3a} t_q[Q_{iw'}]M_j)) \\
&\quad \wedge (t[R_{iw}Q_{iw'}] \neq \varnothing) \\
&\quad \}
\end{aligned}
$$

Case 4

The remaining cases consider joins on nested attributes. This case in

specific considers a join where both nested attributes are at the top level of the relation.

To join nested relations the authors introduce the intersection of two nested relations r and q with the same scheme R. In essence, each resulting tuple exists in both input relations and the sets of tuples of nested attributes are recursively equal as well. In the following definition $N_1, \ldots, N_j$ the set of nested attributes in common, $R_1, \ldots, R_i$ are the set of attributes that are not in common.

$$\begin{aligned} r \cap^{\cap} q \equiv & \{t | (\exists t_r \in r, \exists t_q \in q) \\ & \wedge (t[N_i] = t_r[N_i] \cap t_q[N_i]) \\ & \wedge ((t[R_1] = t_r[R_1] \cap^{\cap} t_q[R_1]) \wedge \ldots \wedge (t[R_i] = t_r[R_i] \cap^{\cap} t_q[R_i]))\} \end{aligned} \tag{Eq. 3.5}$$

The join for relations with nested attributes at the top level can be defined as:

$$\begin{aligned} r \bowtie_4 q \equiv & \{t | (\exists t_r \in r, \exists t_q \in q) \\ & \wedge (t[attr(R) - \{A_1, \ldots, A_j\}] = t_r[attr(R) - \{A_1, \ldots, A_j\}]) \\ & \wedge (t[attr(Q) - \{A_1, \ldots, A_j\}] = t_q[attr(Q) - \{A_1, \ldots, A_j\}]) \\ & \wedge (t[A_1, \ldots, A_j] = (t_r[A_1, \ldots, A_j] \bowtie_4 t_q[A_1, \ldots, A_j]) \\ & \qquad = ((A_{1r} \cap^{\cap} A_{1q}) \wedge \ldots \wedge (A_{jr} \cap^{\cap} A_{jq})) \\ & \} \end{aligned} \tag{Eq. 3.6}$$

Case 5

This case concerns a join in which the two relations have nested attributes in common, of which one is at the top level and the other is not. It is similar to case 2, except for the common attributes being nested instead of atomic. In the following definition relations r and q with join-paths L and M respectively have a set of nested attributes $A_1, \ldots, A_j$ in common. At the end of the join-path, the join is performed according to case 4.

$$\begin{aligned} rL \bowtie_5 qM \equiv & \; r(R_i L_i) \bowtie_5 q(Q_j M_j) \\ \equiv & \; \{t | (\exists t_r \in r) \\ & \wedge (t[attr(R) - \{R_i\}] = t_r[attr(R) - \{R_i\}]) \\ & \wedge ((L_i \neq \varnothing \wedge t[R_i] = (t_r[R_i] L_i \bowtie_5 q) \wedge t[R_i] \neq \varnothing) \\ & \quad \vee (L_i = \varnothing \wedge t[R_i] = (t_r[R_i] \bowtie_4 q) \wedge t[R_i] \neq \varnothing)) \\ & \} \end{aligned} \tag{Eq. 3.7}$$

Case 6a

In this case, the two relations have nested attributes in common, are not at the top, but are at the same depth in the schemes.

In essence this join is equal to case 3a, other than that the attributes in common are nested instead of atomic, i.e. R_i and Q_j are the ancestors of common nested attributes instead of common atomic attributes. When the end of the join paths is reached, the join is performed according to case 4.

$$
\begin{aligned}
rM \bowtie_{6a} qM \equiv\ & r(R_iL_i) \bowtie_{6a} q(Q_jM_j) \qquad \text{(Eq. 3.8)}\\
\equiv\ & \{t|(\exists t_r \in r, \exists t_q \in q)\\
& \wedge (t[attr(R) - \{R_i\}] = t_r[attr(R) - \{R_i\}])\\
& \wedge (t[attr(Q) - \{Q_j\}] = t_q[attr(Q) - \{Q_j\}])\\
& \wedge ((L_i \neq \varnothing \wedge (t[R_iQ_j] = t_r[R_i]L_i \bowtie_{6a} t_q[Q_j]M_j))\\
& \quad \vee (L_i = \varnothing \wedge (t[R_iQ_j] = t_r[R_i] \bowtie_4 t_q[Q_j])))\\
& \wedge (t[R_iQ_j] \neq \varnothing)\\
& \}
\end{aligned}
$$

Case 6b

As the previous three cases, this one is nearly equal to an earlier variant as well. All other definitions being equal, in this case $A_{11}, \ldots, A_{1z}$ consists of nested attributes instead of atomic ones.

$$
\begin{aligned}
rL \bowtie_{6b} qM \equiv\ & r(R_iL_i) \bowtie_{6b} q(Q_jM_j) \qquad \text{(Eq. 3.9)}\\
\equiv\ & \{t|(\exists t_r \in r, \exists t_q \in q)\\
& \wedge (t[attr(Q_{i'}(\ldots(Q_{i+1})))] = t_q[attr(Q_{i'}(\ldots(Q_{i+1})))])\\
& \wedge (t[attr(Q_i) - \{Q_{iw'}\}] = t_q[attr(Q_i) - \{Q_{iw'}\}])\\
& \wedge (t[attr(R_i) - \{R_{iw}\}] = t_r[attr(R_i) - \{R_{iw}\}])\\
& \wedge (t[R_{iw}Q_{iw'}] = (t_r[R_{iw}]L_i \bowtie_{6a} t_q[Q_{iw'}]M_j))\\
& \wedge (t[R_{iw}Q_{iw'}] \neq \varnothing)\\
& \}
\end{aligned}
$$

Although Garani & Johnson have created a clear and concise overview of the nested join operator by creating a distinction between (relative) depth and attribute-type, there is a lot of redundancy in the aforementioned cases.

In order to converge these six cases to a single implementation, as a first step, a generalisation of the nested join is developed below.

By introducing a general definition for attribute equality, the distinction between nested and atomic joins thereby reducing half of the cases. An major advantage of this approach is that a mixture of both nested and atomic attributes can participate in the join operation.

The next definition defines equality of tuples t_r and t_q based on a set of common attributes R, these attributes can be either nested or atomic. $A(R)$ is the set of atomic attributes and $N(R)$ is the set of nested attributes. Furthermore note that, $t[N_i]$ denotes the *set* of child tuples of nested attribute N_i.

$$
\begin{aligned}
t_r \overset{R}{=} t_q \equiv & (\forall A_i \in A(R) : t_r[A_i] = t_q[A_i]) \qquad \text{(Eq. 3.10)}\\
& (\forall N_i \in N(R) : (\forall t_{nr} \in t_r[N_i], \exists t_{nq} : t_{nr} \overset{N_i}{=} t_{nq})\\
& \qquad \wedge (\forall t_{nq} \in t_q[N_i], \exists t_{nr} : t_{nq} \overset{N_i}{=} t_{nr}))
\end{aligned}
$$

Informally equation Eq. 3.10 defines tuple equality as follows. First, all atomic attributes should be equal. Second, all the sets of all nested attributes should be equal, i.e. all tuples in $t_r[N_i]$ should exist in $t_q[N_i]$ and vice versa.

In order to further reduce the number of cases, the definition of the join-path is extended to take into account the absence of a nested relation in the join. This situation occurs when the nodes are not at the same level. In this case, the roots of both input relations are not on the same level in the join result.

In the following definition, the parameter definitions of case 3b are used. Furthermore, $P(R)$ denotes the nested parent attribute of R. A node of the *extended join-path* for a nested join operation with schemes R and Q with total depths i and i' at depth n with common attributes A_c can be defined as:

$$JP_n^{RQ} = \begin{cases} (A_c, A_c), & \text{if } n = 0 \\ (\pi_1(P(JP_{n-1}^{RQ})), \pi_2(P(JP_{n-1}^{RQ}))), & \text{if } n < i \\ (\varnothing, \pi_2(P(JP_{n-1}^{RQ}))), & \text{if } i < n \leq i' \end{cases} \quad \text{(Eq. 3.11)}$$

Again, consider the nested schemes R and Q of case 3b and their parameters. In this generalised case the common attributes $A_c = A_{11}, \ldots, A_{1z}$ can either be nested or atomic. Using equations Eq. 3.10 and Eq. 3.11, the generalised (Garani) join ($\bowtie_G$) of relations r and q can be defined. In the following equation for the sake of brevity, $JP_{nk} = \{\pi_k(JP_{i'-n}^{RQ})\}$.

$$\begin{aligned} r \bowtie_G^{R,Q,n} q \equiv \{t | (\exists t_q \in q) & \quad \text{(Eq. 3.12)} \\ & \wedge (t[Q_n - JP_{n2}] = t_q[Q_n - JP_{n2}]) \\ & \wedge ((n \leq i) \wedge (\exists t_r \in r) \\ & \quad \wedge (t[R_n - JP_{n1}] = t_r[R_n - JP_{n1}]) \\ & \quad \wedge ((0 < n < i) \wedge (t[JP_{n1}] = t_r \bowtie_G^{R,Q,n-1} t_q) \wedge (t[JP_{n1}] \neq \varnothing)) \\ & \quad \vee ((n = 0) \wedge (t_r \overset{JP_n}{=} t_q) \wedge (t[JP_{01}] = t_r[JP_{01}]))) \end{aligned}$$

In this section an extensive overview of the join as proposed by Garani & Johnson [15] was given. In the next section a local algorithm will be presented which implements our generalised version of the G-Join.

3.2 Implementation of the G-join

In the previous section, a generalised version of the Garani-join was introduced in equation Eq. 3.12. In this section an implementation of the generalised Garani-join (G-join) will be presented.

The algorithm starts by finding two equal tuples at the end of the join path, i.e. at the level of the common attributes which is in this case in relations r and q.

Once those are found the parents of the tuples are merged iteratively until the the root of one of the schemes is reached. If one of the tuples still has a parent, that node is added to its parent until the top of the join-path is reached. In this sense the proposed algorithm deviates from definition in equation Eq. 3.12. In this equation, the result nested tuples still have their original siblings. However, checking whether the join within a nested tuple is complete and can hence be shipped, is infeasible in a mediator based environment (see Section 2.5) since the size of the nested structure might well exceed the buffers of the mediator. Therefore, this is considered a post-processing step, independent of mediated join-processing. Under the assumption that the original parent-tuples can be uniquely identified, the join can be post-processed at follows: Starting at the root of the resulting scheme tree, nest underlying relation on the join-path and iterate downwards.

Algorithm 3.1 makes use of Algorithm 2.2, the nested-loop join, to find matching nodes, but it could be implemented using other join algorithms as well. It uses Algorithm 2.1 to check for tuple equality. Without loss of generality this algorithm assumes that the depth of Q is greater than or equal to the depth of R.

Algorithm 3.1 An implementation of the G-join on nested relations r and q.

```
1:  result := {}
2:  for all t_r ∈ r do
3:    for all t_q ∈ q do
4:      if t_r = t_q then
5:        i := 0
6:        t := t_r
7:        while P(t_r) ∧ P(t_q) do
8:          t_r := P(t_r)
9:          t_q := P(t_q)
10:         i := i + 1
11:         t_tmp[R_i − {π_1(JP_i)}] := t_r[R_i − {π_1(JP_i)}]
12:         t_tmp[Q_i − {π_2(JP_i)}] := t_q[Q_i − {π_2(JP_i)}]
13:         t_tmp[{π_1(JP_i)π_2(JP_i)}] := t
14:         t := t_tmp
15:       end while
16:       while P(t_q) do
17:         t_q := P(t_q)
18:         i := i + 1
19:         t_tmp[Q_i − {π_2(JP_i)}] = t_q[Q_i − {π_2(JP_i)}]
20:         t_tmp[{π_2(JP_i)}] := t
21:         t := t_tmp
22:       end while
23:       result := result ∪ {t}
24:     end if
25:   end for
26: end for
```

This algorithm will provide the basis for the implementation of the generalised G-join. In this chapter an analysis of the nested relational join was presented,

concluding with a generalised implementation. In the next chapter, the experiment set-up is discussed.

Chapter 4

Experimental set-up

In this section the experimental set-up is described. First the experiment design is presented in Section 4.1. Second, assumptions that underlie this experiment are stated in Section 4.2. Subsequently, the input data are discussed in Section 4.3. After that the parameters are presented in Section 4.4. The implementation of the experiment is discussed in more detail in Section 4.5.

4.1 Experiment design

In this section the overall design of the experiment will be described. The experiment has several input parameters, each of which has several values. A particular instance of the experiment is created when a value is chosen for each of these parameter values: this is a called a *configuration*. A complete set of configurations is created by taking the Cartesian product of all parameters. The parameters and their actual values are outlined in Section 4.4. The experiment is complete after each of these configurations has been executed.

For each configuration, a *Client* is instantiated which has a certain information need. This information need is interpreted by the *Mediator* which processes the query and returns the *Query result* to the client. During this process several events are logged in order to derive *Statistics* of the run. The query result is only used for testing purposes, but it is discarded during final execution of the experiment since the only output of interest are the output parameters.

An overview of the set-up is depicted in Figure 4.1.

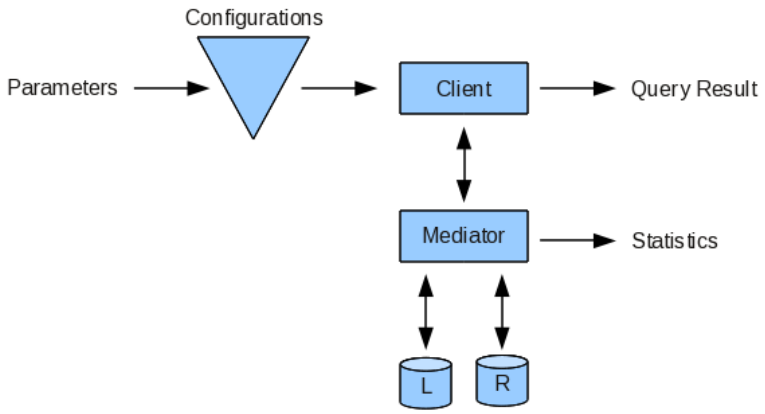


Figure 4.1: *The experimental set-up*

4.2 Assumptions

In this section the assumptions underlying this experiment are presented. These assumptions were made to frame this experiment. In Chapter 5 the implications of relaxing each of these assumptions are discussed.

A1. *The storage of meta data does not impact the buffer.* In order to operate over the different databases, the mediator needs to have meta data of the participating data sources. Because the amount of meta data to store is small (typically a few kB) compared to the amount of data that is joined it is considered to be part of the internals of the Medaitor.

A2. *The mediator can only be queried for a join operation*, it does not support other relational operators. In order to limit the scope of the experiment, the number of commands understood by the mediator is limited to the one this study is all about.

A3. *The client is interested in the complete join result.* Following from the previous assumption, the client is only interested in the complete join result.

A4. *The data format used by the client is equal to the one used by the mediator.* Since the client could be accepting input in any format, the format of the mediator was chosen as the target format. The advantage of this is that output data can be handled using the same tools as the input data, which eases output validation.

A5. *The client is indifferent about the order in which partial results are sent.* Ordering is still a complex operation. To order a result set in a mediator-based context would mean keeping the complete result in memory. This is simply infeasible for larger relations.

A6. *To mitigate start-up costs, there is no query cache available between client queries.* During query execution results can be stored, at the cost of buffer

space.

A7. *The data on the sites are stored using a decomposed storage model.* The decomposed storage model is an approach to store nested data in a set of non-nested relational database, by splitting up every relation.

A8. *All sites have the capability to sort tuples of a relation according to a specific (set of) attributes(s).* Since the sort-merge join requires ordering of the results, sites need to have this capability. All mainstream database solutions support ordering of a result set.

A9. *All sites have the capability to perform selection (σ) and projection (π) on relations.* In order to select blocks tuples from the databases, support of the selection operator is required. Next to that, in order to use the semijoin communication strategy (see Section 4.4) the projection operator is required. All mainstream DBMS support the projection and selection operators.

A10. *The minimal amount of memory available in the mediator is sufficient to process at least one complete tuple at a time.* If the mediator does not have this minimal amount of memory, it becomes impossible to join information. Depending on the scheme of the relation, this lower bound will typically be around a few kB.

In this section the assumptions under which this study is conducted were presented. In the next section, the input data of the experiment and the derivation thereof is presented.

4.3 Input data

As mentioned before, the input data are of a hierarchical nature. In order to control the output of the experiment, different data sets of different sizes, with a set of common attributes were required. Next to that, the result of the application of the nested join operator needed to be known in advance so runs could be designed based on certain parameter values (discussed in Section 4.4).

Since readily available data-sets and fake-data generators [6] were not flexible enough to generate data that satisfied the needs of this experiment two highly configurable data generation tools were created.

The first tool was a data generation tool, nicknamed `TreeGen`. The second tool was concerned with data duplication in order to be able to create a join result which would adhere to predefined parameter values. This tool was nicknamed `TreeDup`.

The `TreeGen` program requires a XML-based configuration file as its input which describes the scheme tree of the nested data, consisting of relations and attributes. An example of such a configuration file can be found in appendix A. The program generates a SQLite[1] database where the nested data are stored

[1] See http://www.sqlite.org

using a decomposed storage model (DSM). The DSM is implemented in such a way that each tuple has a unique identifier, and each tuple of a nested attribute has a reference to its parent tuple in order to facilitate tree traversal when the G-Join is executed. In essence, the `TreeGen` process consists of the following steps:

1. Read the configuration file.
2. Create a temporary database in memory.
3. Create a data structure according to the DSM.
4. Generate of non-referenced data (i.e. the literal values).
5. Generate of referenced data (i.e. parent-child relations and foreign keys).
6. Store the database on disk.

`TreeDup` selects a (nested) part of one dataset and replaces it into another. As such, it requires three input parameters: a scheme which is a modified version of the previous XML-configuration file, a source-database and a target-database. The tree duplication process is the following:

1. Read the configuration file.
2. Fetch a predefined amount of nested tuples from the source database, these tuples will be duplicated.
3. Delete the same amount of tuples in the target relation.
4. Store the fetched tuples in the target database.
5. Fix all references.

In this section the process of generating input data for the experiment has been presented. In the next section the parameters related to the experiment will be discussed.

4.4 Parameters

As mentioned in Section 4.1 this experiment has a number of input parameters. The input parameters were chosen both to mimic various environmental conditions and to study the effect of several structural changes of the experiment. The input parameters are:

1. Join algorithm
2. Communication strategy
3. Left and right database size
4. Success ratio
5. Success ratio cardinality

6. Buffer factor

First of all, the join strategy is one of two algorithms introduced in Chapter 2; the nested-loop join (Algorithm 2.2) or the sort-merge join (Algorithm 2.3). The communication strategy is one of the strategies defined in Section 2.4: naive or semijoin.

The left and right databases are of different sizes: $S = \{1, 10, 100\}$. Sizes of the databases are defined as the number tuples of the root node. The set of sizes for the left and right hand side of the join can be defined as:

$$\{(s_L, s_R) | s_L \in S, s_R \in S \wedge s_L \geq s_R\}$$

This means that the databases on the left hand side of the join are greater than or equal to those of the right hand side.

The *success ratio* and the *success ratio cardinality* parameters are used to describe the properties of the result set of the join. The success ratio is defined as the percentage of distinct tuples of the smallest input relation that occur in the other input relation. By defining the success ratio based on thirds, a large variety of overlap between the two datasets can be simulated.

The success ratio cardinality was introduced to denote that multiple matches can occur for the same key. It is defined as the percentage of distinct matching tuples in the result set over the total number of tuples in the result set. In selecting the parameter values the notion of a match occurring multiple times needed to be varied. In order to make a clear distinction between these occurrence counts, four subsequent values each differing a factor of ten were chosen.

Since the mediator has a fixed memory capacity, it uses buffers internally to store temporary results and input data in order process queries. To account for this capacity, a *buffer factor* was introduced. At a minimum the mediator should be able to process a single tuple at a time, in other configurations, the mediator should be able to keep a certain amount of the data in memory. This percentage is defined as a power of this one tuple memory constraint. The values for these buffer factors were the bare minimum (0), a quarter of the memory needed and half of the memory needed. A larger buffer exponent would not mimic realistic behaviour, i.e. as the amount of memory increases, complete relations can be kept in memory. This requires major modifications to the algorithms used and was therefore considered to be out of scope.

In a run, several variables are captured. First of all the number of bytes sent between mediator and client and between mediator and database over different phases is recorded. Second the duration of several activities is logged. These data are logged to a file.

In Table 4.1 an overview of the parameters and their values is presented.

The output parameters are: the duration and the number of bytes sent. These are sensible output parameters since they represent the speed of the operation and the operational cost.

Table 4.1: *An overview of the input parameters.*

Input parameter	values
Join strategy	$\{NLJ, SMJ\}$
Communication strategy	$\{Naive, Semi\text{-}join\}$
Left and right database size	$\{(s_L, s_R) \mid s_L \in S, s_R \in S \wedge s_L \geq s_R\}$
Success ratio	$\{0.0, 0.33, 0.67, 1.0\}$
Success ratio cardinality	$\{0.1, 1.0, 10.0, 100.0\}$
Buffer factor	$\{0, .25, .5\}$

In this section an overview of the parameters used in the experiment was given. In the next section the actual implementation of the experiment is discussed.

4.5 Implementation

In this section the implementation of the experiment is discussed in more detail. An overview of the implementation showing all major components is depicted in Figure 4.2.

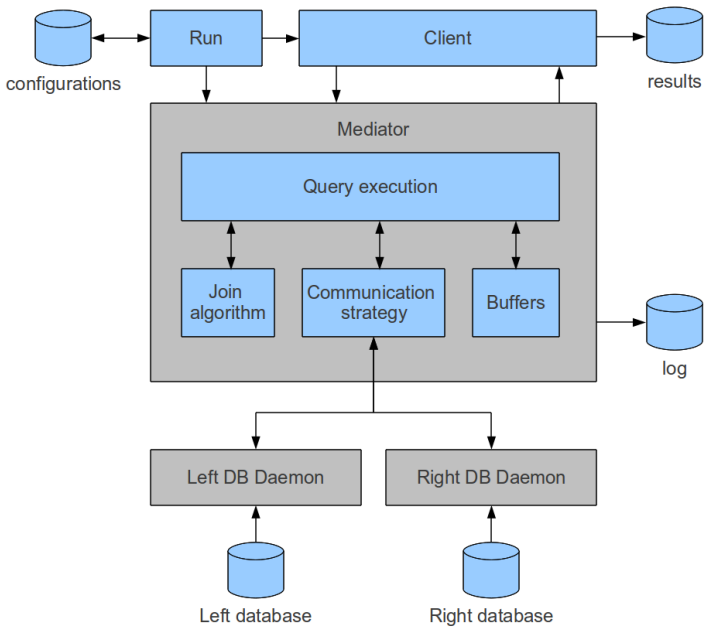


Figure 4.2: *Overview of the implementation of the experiment*

First a random configuration which has not yet been executed is selected from the set of experiments. Based on the configuration, the left and right databases are loaded into main memory for efficiency reasons and allocated to two separate threads. A multi-threaded set-up was chosen to mimic the mediator and its environment. By doing so, databases can be queried simultaneously.

Once the databases are loaded in main memory, a mediator instance is created with the parameters of the current configuration. After this initialisation phase the mediator receives a user query, which is executed by using a join algorithm (Algorithms 2.2 and 2.3) together with a communication strategy (Section 2.4). The communication strategy handles the communication with the left and right databases through their daemons.

As the join executes, several events are logged to an output stream. The output format of these events are tab separated values, in order to be able to insert these logs into a database for analysis later on.

The algorithms are implemented using standard C++[2], using the LibXML2[3]

[2] Adhering to the 1998 ANSI/ISO C++ standard.
[3] See http://xmlsoft.org

library for parsing the configuration/metadata files. The configuration, input-data, as well as the run results are generated using SQLite[4].

Timing of durations is done using `clock_gettime`[5] defined in `<time.h>` using `CLOCK_MONOTONIC` since the system on which the code is deployed does not guarantee that processes will not be migrated to another CPU.

Because of its memory requirements, the final code is deployed on the Mammoth server of the TU/e. This is a computational system which consists of 7 servers that appear as one. It has 56 2GHz processors, a total of 935GB of memory, and a local disk of 2.4TB. This system runs a modified 64-bit Fedora 12 distribution. The Linux command `taskset`[6] is used to specify that the CPU's which execute the program were limited to one of the 7 underlying servers.

This section concludes the description of the experimental set-up. In the next chapter the results of the experiment will be presented.

[4] Version 3.7.2, see http://sqlite.org
[5] See the Linux User's Manual, `man clock_gettime`
[6] See the Linux User's Manual, `man taskset`

Chapter 5

Experiment evaluation

In the previous chapter, the experimental set-up was presented. In this chapter the results of the experiment are presented and discussed.

This chapter consists of four sections: In the first section the duration of each of the experiment runs is analysed. The second section concerns the amount of data send for each of the experiments. Next, the combination of run size and duration is inspected in Section 5.3. Finally, the most important observations will be sumarized in Section 5.4. The initial experiment parameter values as presented in Section 4.4 and 'logical' combinations thereof form the basis of the evaluation.

Note:

The continuous lines used in the graphs do not imply that the output is continuous. This style was merely chosen for clarity, i.e. to indicate the change in value. Next to that, the top part of the error bars in the graphs denotes the standard deviation of the average value, while the bottom part denotes the minimum value.

5.1 Duration of Runs

In this section the duration of runs is compared to input parameter values. The run duration was determined using the commands described in Section 4.5. Measurement of a run started after both database threads had fully loaded their database in memory. Each run was executed 5 times in order to reduce timing fluctuations occurring due to system usage by other processes and/or users.

In Figure 5.1 the average duration runs and their standard deviation is depicted for each of the algorithm – join-strategy combinations. In this figure NLJ and SMJ are acronyms for the Nested Loop Join and the Sort Merge Join defined in

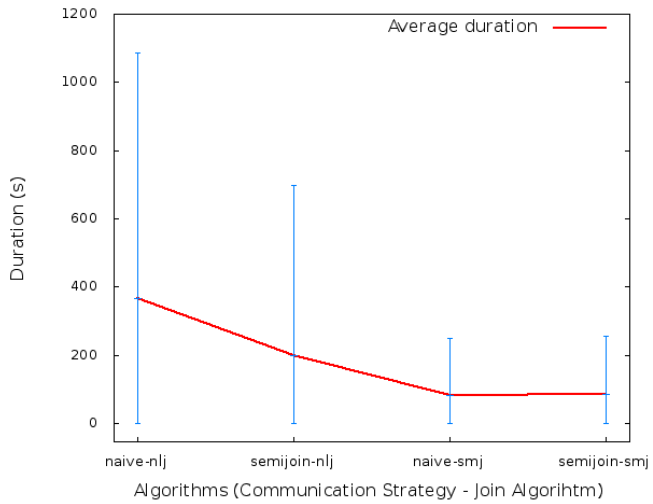


Figure 5.1: *The duration of runs for the algorithms and join-strategies.*

Section 2.3, the Naive and Semijoin labels refer to the communication strategies defined in Section 2.4.

From this figure, it can be seen that on average the sort-merge join as described in Algorithm 2.3 outperforms the nested loop join (Algorithm 2.2). When comparing the performance of the naive join- strategy and the semijoin there is a difference when the nested loop join is used, however there is no apparent difference when using the sort-merge join. This can be explained by considering the operations at a lower operational level. The NLJ iterates over its outer and inner relations. Since these relations are not ordered, each outer block is compared to each inner block. By using a semijoin approach inner tuples which do not participate in the join are not shipped and checked. Thereby the amount of tuples is reduced which results in a lower run duration.

For the sort-merge-join on the other hand, the difference is less notable, since results were ordered, a smaller set of tuples is compared. However, since less tuples are sent to the mediator it can be concluded that the actual comparison of tuples and the complexity of the join algorithm is an important factor in the execution of the nested join.

Next, the size of the relations involved in the nested join is analysed. In Figure 5.2 the duration is plotted as a function of the left and right database sizes. Note that the y-axis of the graph uses a logarithmic scale. This means that the complexity of the join-algorithm increases in a non-linear manner when the size of the participating relations increases. The dip in the graph is likely to be caused by difference in complexity between the $100-1$ and the $10-10$ instances with respect to the communication strategy and join-algorithm used. The semijoin combined with the nested loop join needs less iterations in case of $100-1$, hence resulting in a lower overall duration. This expectation is confirmed by breaking down Figure 5.2 over the different communication strategies and join algorithms. This breakdown is depicted in Figure 5.3.

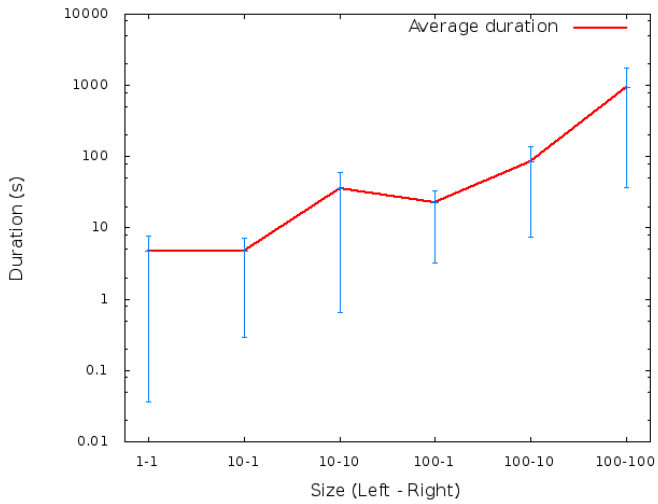


Figure 5.2: *The duration of runs for left and right join sizes.*

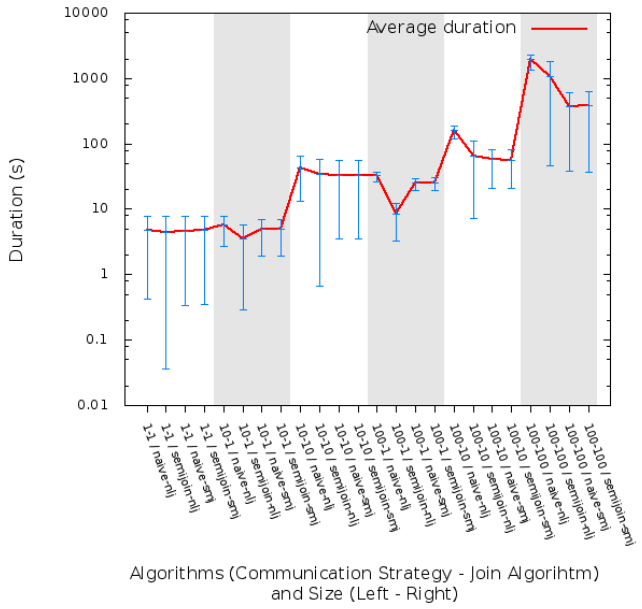


Figure 5.3: *The duration of runs for left and right join sizes broken down by the different communication strategies and join algorithms.*

The success ratio determines the amount of matching tuples between the two relations. Figure 5.4 depicts the different parameter values and the associated average run times. As can be seen from this figure, a higher success ratio results in more matching tuples, which in turn results in longer processing time, with a larger degree of variability.

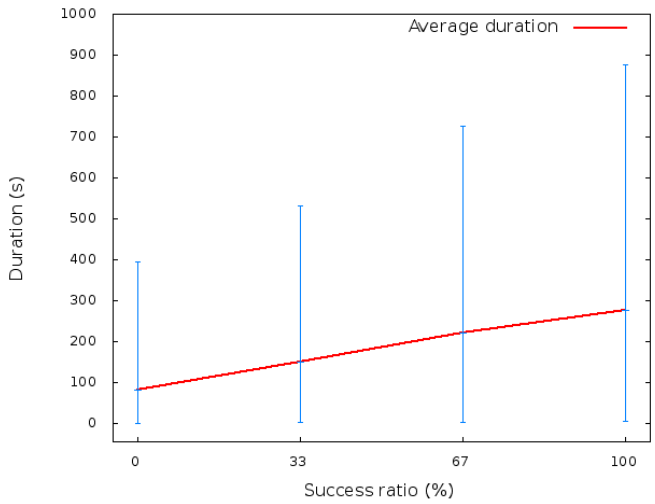


Figure 5.4: *The duration of runs for different success ratio values.*

The success cardinality describes the percentage of similar matches between two relations. The average run durations for the different values of this parameter are depicted in Figure 5.5. From this figure, it can be seen that the average run

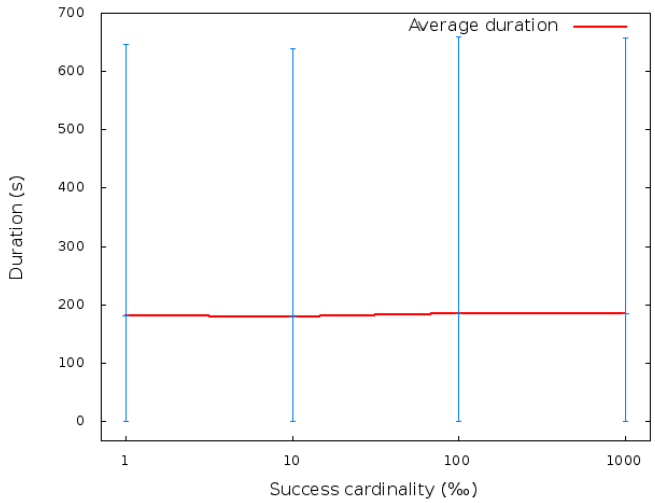


Figure 5.5: *The duration of runs for different success cardinality values.*

durations are hardly influenced by this parameter. To confirm this hypothesis, all combinations of success ratio and success cardinality are compared to the run duration in Figure 5.6. As expected, the run size only increases with different success ratio's and is again indifferent for the success cardinality. This makes sense since the amount of tuples matched stays the same due to this parameter. Because success cardinality is introduced in the data duplication step, tuples are duplicated in the right relation only, resulting in a 1–N relation. Therefore the

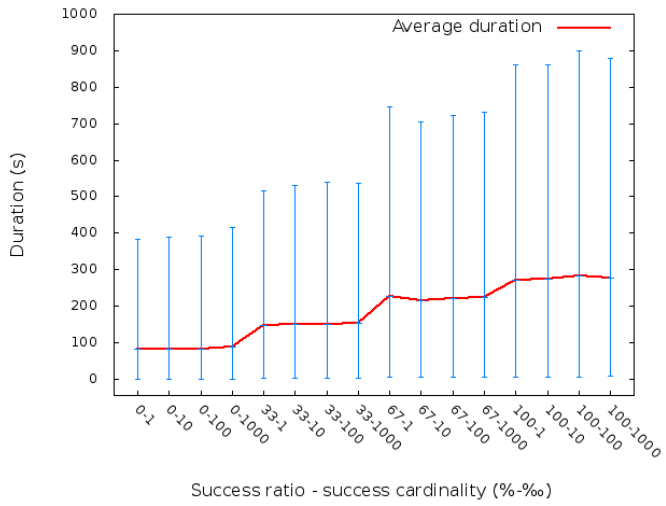


Figure 5.6: *The duration of runs for different success ratio and success cardinality values.*

benefits one would normally get from using a semijoin communication strategy do not show up here because the same amount of tuples is matched.

Leaving us with the buffer exponent parameter. Figure 5.7 depicts the durations over the different buffer exponent parameter values. Figure 5.7 shows that this

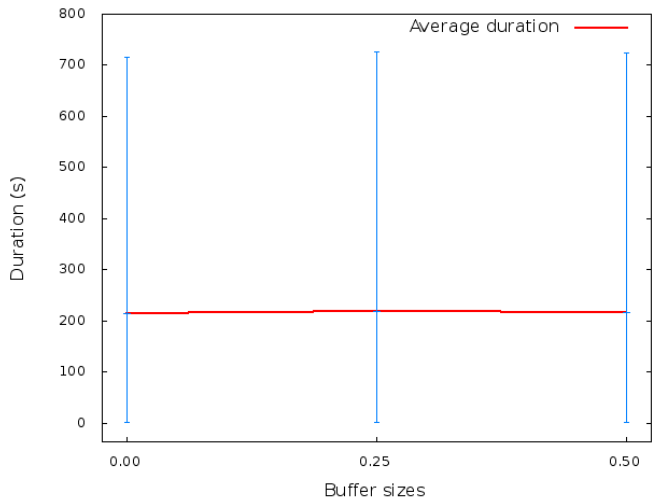


Figure 5.7: *The duration of runs for different buffer exponent values.*

parameter has hardly any effect on the run duration.

In this section an analysis of the duration of the different experiment runs was made. In the next section an analysis of the number of bytes sent is discussed.

5.2 Number of bytes sent during runs

In this section the number of bytes sent is compared to different input parameter settings. The amount of bytes sent consists of the number of bytes sent between both databases and the mediator, and the number of bytes sent between the mediator and the client.

The amount of data sent for each combination of communication strategy and join-strategy is depicted in Figure 5.8. There is practically no difference between

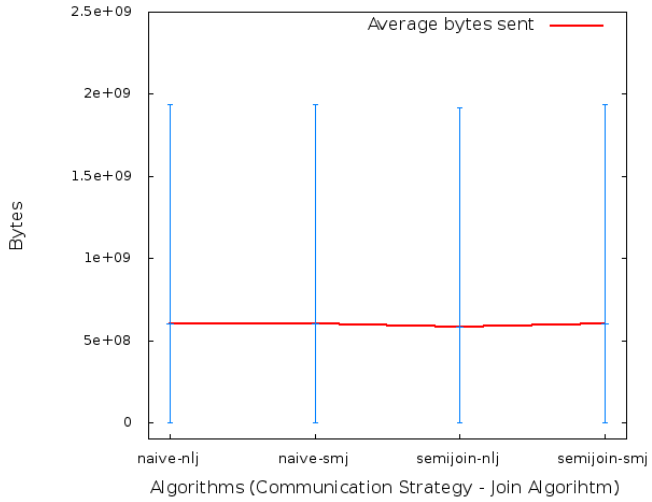


Figure 5.8: *The number of bytes sent for the algorithms and join-strategies.*

the number of bytes sent for both the naive and the semijoin SMJ which is unexpected when looking at the joins at a lower level. One explanation for this behaviour is the usage of blocked verions of the join algorithms which affect the effectiveness of communication strategies.

Figure 5.9 indicates the number of bytes sent over different parameter left and right database sizes. Note that the scale of the y-axis is logarithmic. The average number of bytes is lower for the $100 - 1$ size combination. This is likely to be caused by the semijoin algorithm, which reduces the amount of data required to be sent to the mediator, in combination with the ratio of the right hand site over the left hand side.

In Figures 5.10 and 5.11 the average total number of bytes sent is depicted for the success ratio and the success ratio cardinality respectively. As is to be expected, the success ratio affects the total number of tuples sent. When this parameter equals 0, no tuples match and hence no join result exists. Therefore, the client receives an empty result set and there is only communication between the two databases and the mediator. The success cardinality does not affect the average total number of bytes sent, since the amount of matches does not change with this parameters. Again, Figure 5.12 confirms this.

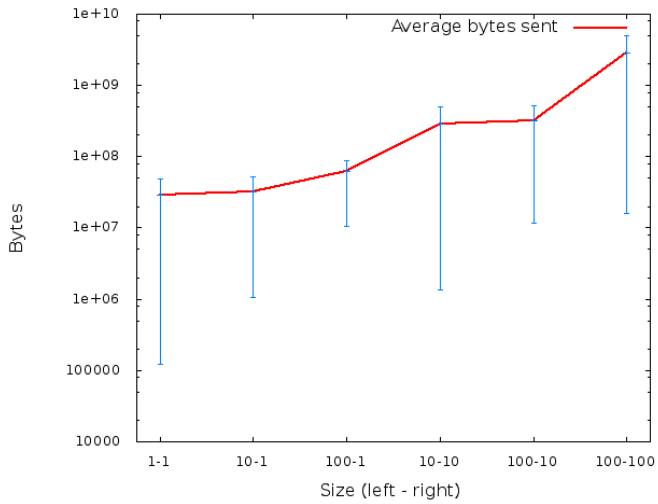


Figure 5.9: *The average number of bytes sent for the left and right join sizes.*

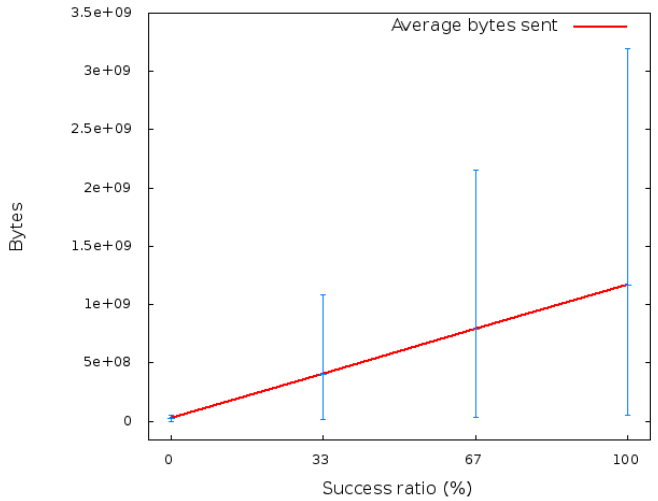


Figure 5.10: *The average total number of bytes sent for different success ratio values.*

Finally, the buffer size does not affect the total number of tuples sent, as is depicted in Figure 5.13.

In this section an overview of the different parameters and their effect on the average total number of bytes sent during a set of experiment runs was presented. In the next section the run duration is compared to the run size.

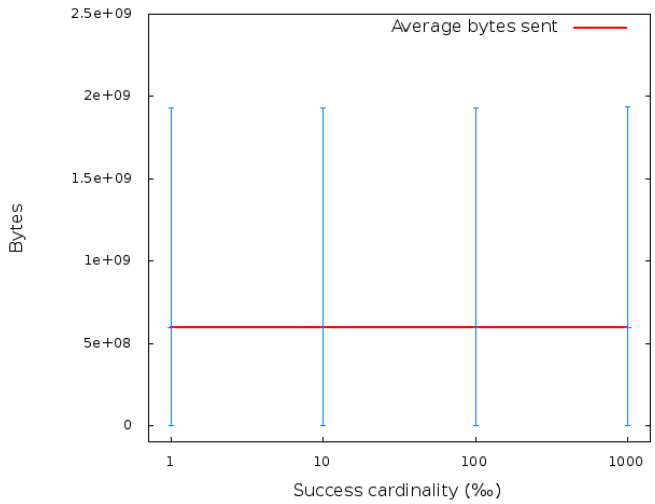


Figure 5.11: *The average total number of bytes sent for different success cardinality values.*

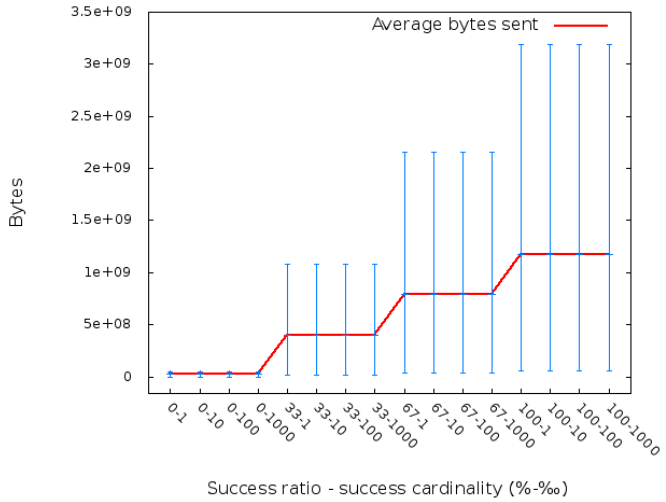


Figure 5.12: *The average total number of bytes sent for different success ratio and success cardinality values.*

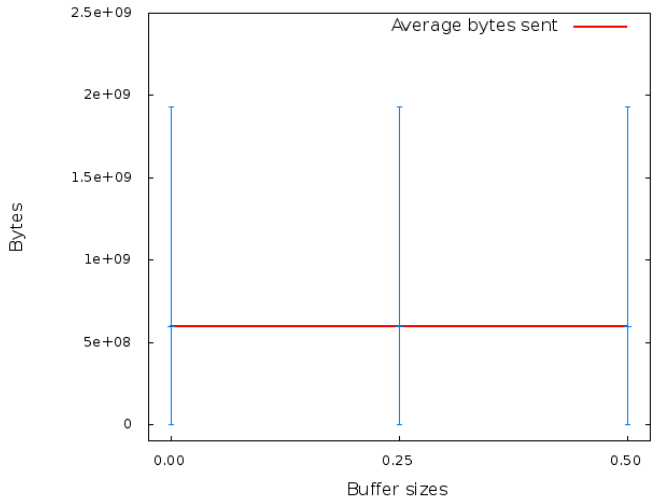


Figure 5.13: *The average total number of bytes sent for different buffer exponent values.*

5.3 Run duration vs. number of bytes sent

In the previous two sections duration and the number of bytes sent were offset by the input parameters and combinations thereof. In this section the relation between both output parameters is inspected.

In Figure 5.14 the duration is plotted against the number of bytes sent. In order to clarify this graph, colour coding is added based on the input size of left and right relations participating in the join. A single point in this graph represents a single experiment run. In total, 5760 runs were performed, 5 for each individual run configuration. The colour coding clearly shows that the

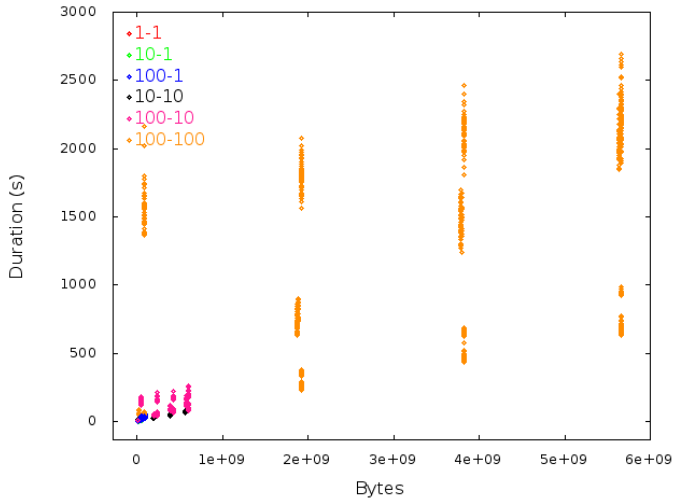


Figure 5.14: *The run duration offset by the number of bytes sent colour coded by input sizes.*

relation size of both databases predicts both run duration and the number of bytes sent. As a result, the graph contains repetition for each major change in size. A more detailed figure which magnifies the area were the repetition occurs can be found in Appendix B, page 1.

Within the each of these magnifications, clear groups of runs can be distinguished along the horizontal axis. These groups are formed due to the different values of the success ratio parameter, as depicted in Figure 5.15. From this figure it is clear that the higher the join ratio, the more bytes have been sent during the experiment. Magnifications of the lower left corner of the figure can be found in Appendix B, page 2. With this graph the experimental set-up is verified: the join-ratio determines the size of the join result based on the smallest input relation.

For each of the groups formed by the ratio parameter depicted in Figure 5.15, subgroups can be identified along the vertical axis. These subgroups are formed, albeit with overlap, by the communication strategy and the join algorithm. This is clarified by Figure 5.16. Again, magnifications of the lower left corner can be

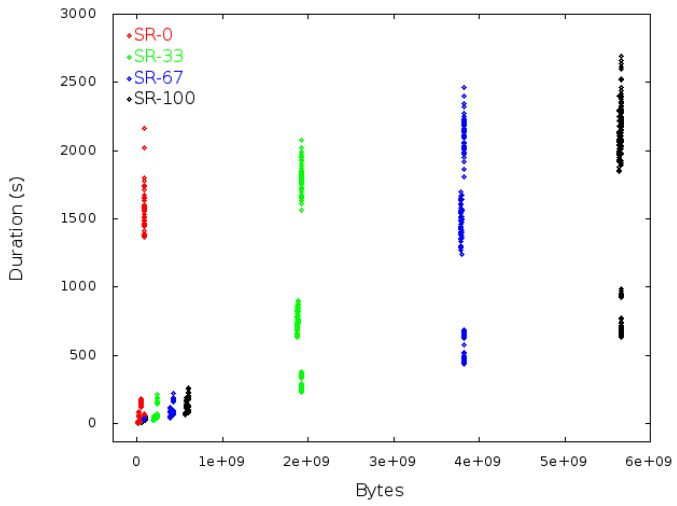


Figure 5.15: *The run duration offset by the number of bytes sent colour coded by success ratio.*

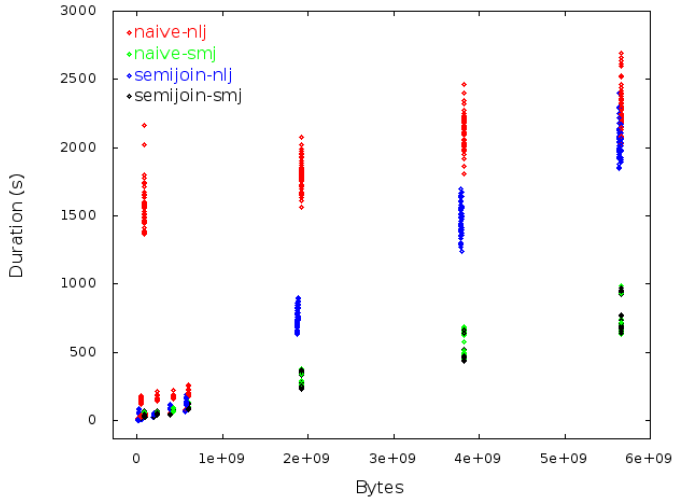


Figure 5.16: *The run duration plotted against the number of bytes sent colour coded by the combination of the communication strategy and the join algorithm.*

found in Appendix B, page 3. Note that the colour coding of this figure over all magnifications corresponds to the findings in Figure 5.1, i.e. usage of the sort merge join results in a lower run duration.

In this section the two output parameters, duration and the number of bytes sent, were analysed. In the next section a summary of this chapter is given.

5.4 Summary

In the previous sections the duration and number of bytes used for each experiment run were analysed. First the run duration and the run size were inspected. After that, the output parameters were analysed. Over all runs, the following properties were noted:

- The usage of the sort merge join algorithm reduces the run duration (Figure 5.1 and Figure 5.16).
- Using semijoin as a communication strategy only has an effect on run duration when not used in combination with the sort merge join (Figure 5.1).
- The combination of input sizes of the left and right relations affect both the number of bytes sent and the run duration (Figure 5.2, Figure 5.9 and Figure 5.14).
- The success ratio determines the number of matches of the join operation. Therefore, by design, the success ratio affects the number of bytes sent (Figure 5.10). However it also has a clear, predictable impact on the duration of the join (Figure 5.4).
- The experiment, although subject to a wide range of configurations, performs in a predictable manner. This is shown by the repetition which occurs in the graphs where the duration is offset by the number of sent bytes (Figure 5.14 and Appendix B).

In this chapter the duration and number of bytes used for each experiment run were discussed and several observations were made. The next chapter presents the conclusions of this thesis.

Chapter 6

Conclusions

This chapter presents the conclusions of this thesis. The subject for this thesis evolved out of the absence of a nested relational join operator required to join hierarchical data in a distributed context, originally extending the Data Flow Language as introduced by Hidders *et al.* [16].

Several extensions were proposed [11, 18, 24] however all with limitations. Garani & Johnson [15] were the first to present a clear overview of all cases in which one might perform a nested-join, providing the set-up towards a uniform nested-relational operator that is not hindered by any schematic limitations. The collection of these cases was dubbed the G-Join.

After careful analysis of these six cases of the G-Join, a generalized version was designed in Chapter 3. This generalization answered the first sub research question, i.e. it described the output of a nested-relational join.

This generalized version was used as a set-up for a single algorithm which was later on extended to a distributed, mediator-based context. The implementation of the algorithm and the algorithm itself were evaluated in an experiment subject to several parameters. The experimental set-up was described in Chapter 4 and the evaluation of the experiment in Chapter 5.

6.1 Contributions

For completeness the contributions as mentioned in Section 1.4 are stated here. The contributions of this thesis consist of:

- The design of a generalized version of the G-Join (Section 3.1).
- Design of an algorithm based on this generalized version (Section 3.2).
- An implementation of a nested-relational join operator in a framework (Chapter 4)

- A first empirical evaluation of the generalized G-Join. (Chapter 5).

6.2 Experimental findings

In this section the experimental findings are described based on Chapter 5. Next to that, the usefulness of the input parameters is assessed and the third sub research question is answered.

The analysis of the output parameters yielded the following findings:

- The usage of the sort merge join algorithm reduces the run duration (Figure 5.1 and Figure 5.16).
- Using semijoin as a communication strategy only has an effect on run duration when not used in combination with the sort merge join (Figure 5.1).
- The combination of input sizes of the left and right relations affect both the number of bytes sent and the run duration (Figure 5.2, Figure 5.9 and Figure 5.14).
- The success ratio determines the number of matches of the join operation. Therefore, by design, the success ratio affects the number of bytes sent (Figure 5.10). However it also has a clear, predictable impact on the duration of the join (Figure 5.4).
- The experiment, although subject to a wide range of configurations, performs in a predictable manner. This is shown by the repetition which occurs in the graphs where the duration is offset by the number of sent bytes (Figure 5.14 and Appendix B).

Usage of the success ratio cardinality parameter did not yield any relevant data. This is partly due to the way the success ratio cardinality was implemented; as a 1–N relation. Although not sure, it can be questioned whether a N–N implementation would have turned up real differences. The success ratio parameter would need to be redefined in order to keep it reliable. Also the buffer size did not affect duration and the number of bytes sent a significantly.

Except from the latter two parameters, the rest of the input parameters returned one or more insights. Within the context of this research this is the answer to sub research question 3. Since the possible input parameters which could be used to extend this experiment are plentiful, this set is likely to grow in further research.

In this section the experimental findings were presented and the third research sub-question was answered. In the next sections the limitations and future work are discussed.

6.3 Limitations and future work

This section was modified since it contained classified information.

The contributions as mentioned in Section 6.1 are only the beginning of the large number of challenges spanning a number of research areas that will need to work before a nested relational operator can be considered operational.

Dropping assumptions A2, the mediator can only be queried for the join operation, and A3, the client is interested in the complete join result, are logical next steps towards this goal. By dropping these assumptions more advanced queries can be requested from the mediator.

A limitation of this study is the absence of a post-processing step which is used to nest all found tuples in their original hierarchy. As mentioned in Section 3.2 this was considered infeasible due to the very nature of the context. A separate study on a new communication strategy might resolve this issue. In the current approach a lot of nested tuples which are sent to the client are in fact redundant. Creating an improved communication strategy which resolves this would be a huge win.

All in all, although this thesis has provided insight into several aspects, it has raised even more follow-up questions. This is a good sign.

In this section the limitations and the future work were addressed. In the next section the research question will be answered.

6.4 Research question

After discussing the limitations the second sub research question has been answered with a partial 'Yes'. The implementation and the evaluation showed a successful implementation of a nested relational join operator in a mediator-based distributed environment. However, due to the nature of the mediator, composition of the resulting tuples had to sacrificed.

Leaving us with the research question: ***How can the nested relational join operator be implemented in a mediator-based distributed environment?***

This question is answered a follows:

The nested relational join operator can be implemented in a mediator-based distributed environment by using the generalized version of the G-Join as presented in Chapter 3, the context of the mediator however poses significant challenges with respect to the composition of the nested results. On this specific subject, further research is needed. Other properties that affect the performance of this operator include: the left and right relation sizes, the communication strategy, the join algorithm and the success ratio.

Appendix A

TreeGen configuration file

In this appendix, an example configuration file as used by the `TreeGen` program is described. In this configuration file, (nested) relations and attributes along with their properties can be described using a simple XML syntax.

```
<config>
  <!-- The file the database is written to -->
  <output_file>supplier.db</output_file>

  <!-- The schema -->
  <relation name="Supplier" count="1">
    <attribute name="id" type="pk" />
    <attribute name="name" type="string" />
    <relation name="BankAccount" count="5">
      <attribute name="account_number" type="string" />
    </relation><!-- /BankAccount -->
    <relation name="SupplierLocation" count="3">
      <attribute name="id" type="pk" />
      <attribute name="name" type="string" />
      <attribute name="address" type="string" />
      <attribute name="country" type="string" />
      <attribute name="phone" type="string" />
      <relation name="Article" count="500">
        <attribute name="id" type="pk" />
        <attribute name="code" type="string" />
        <attribute name="name" type="string" />
        <attribute name="description" type="string" />
        <attribute name="price" type="float" min="1" max="120" />
        <attribute name="price_group_id" ref_rel="PriceGroup"
          ref_constraint="same_parent" ref_attr="id" />
        <attribute name="category_id" ref_rel="Category" ref_attr="id"
          />
        <relation name="ArticleRevision" count="5">
          <attribute name="id" type="pk" />
          <attribute name="code" type="string" />
          <attribute name="date" type="date" date_from="1990-01-01" />
          <attribute name="price" type="float" min="1" max="120" />
        </relation><!-- ArticleRevision -->
      </relation><!-- /Article -->
    </relation><!-- /SupplierLocation -->
    <relation name="PriceGroup" count="50">
      <attribute name="id" type="pk" />
```

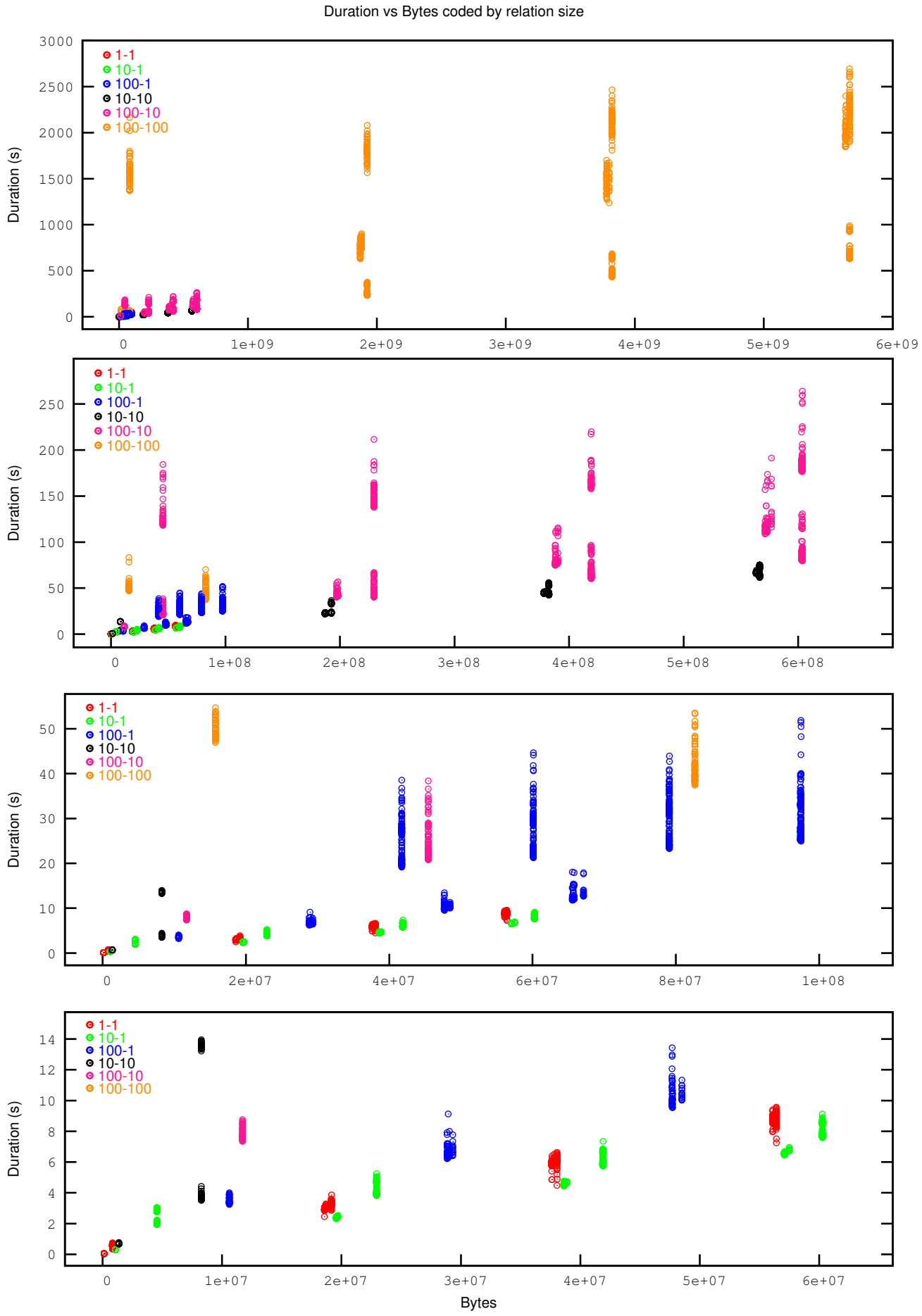
```
      <attribute name="name" type="string" />
      <relation name="BulkQuantities" count="4">
        <attribute name="quantity" type="int" min="1" max="10000" />
        <attribute name="discount" type="int" min="5" max="15" />
      </relation><!-- /BulkQuantities -->
    </relation><!-- /PriceGroup -->
  </relation><!-- /Supplier -->
  <relation name="Category" count="50">
    <attribute name="id" type="pk" />
    <attribute name="name" type="string" />
    <attribute name="parent_id" ref_rel="Category" ref_attr="id"
      null_prob="0.5" />
  </relation><!-- /Category -->
  <relation name="Warehouse" count="4">
    <attribute name="name" type="string" />
    <attribute name="address" type="string" />
    <relation name="Inventory" count="3000">
      <attribute name="article_id" ref_rel="Article" ref_attr="id" />
      <attribute name="stock_level" type="int" dist="uniform" min="-10000"
        max="300000" />
    </relation><!-- /Inventory -->
  </relation><!-- /Warehouse -->
</config>
```

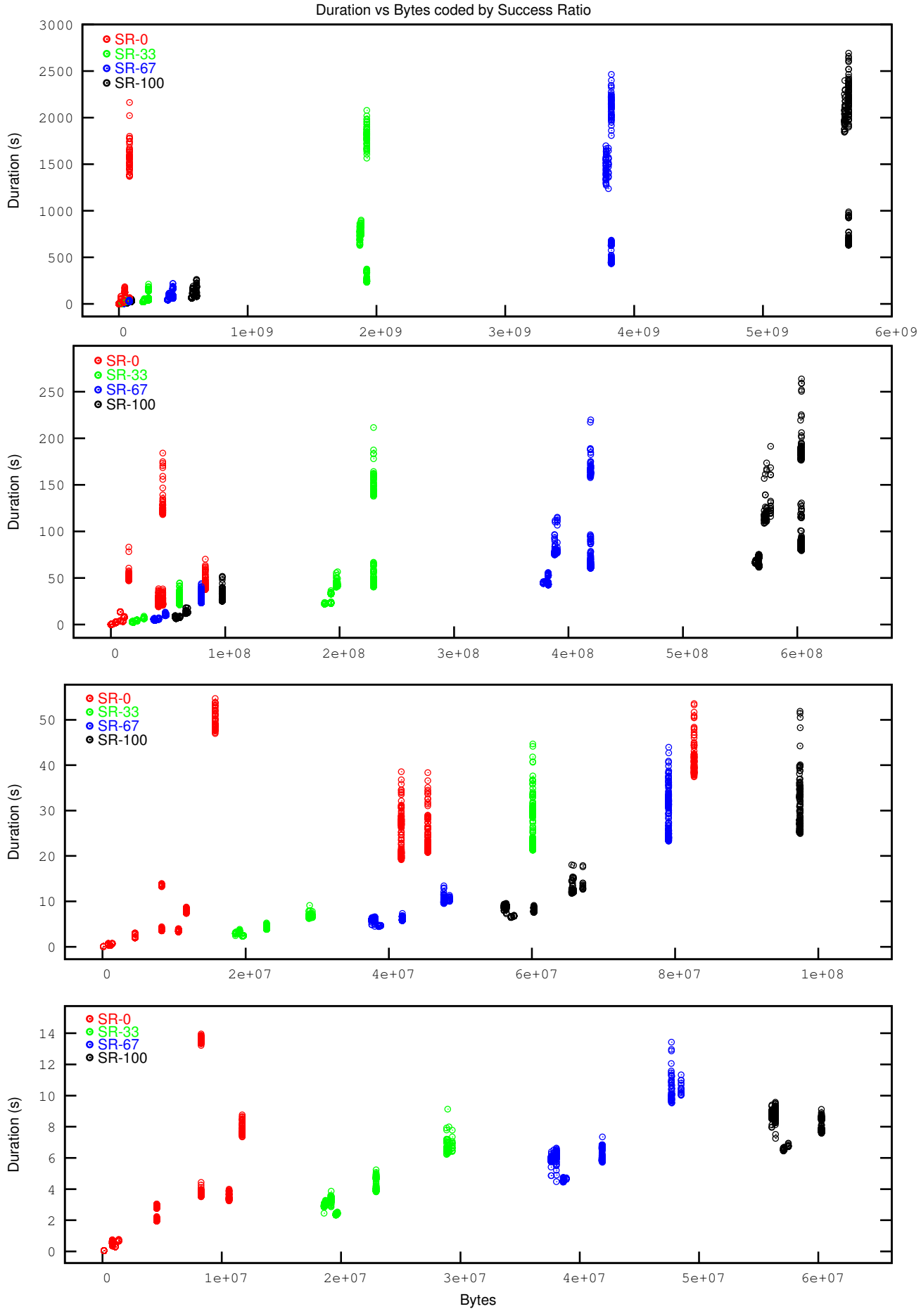
Appendix B

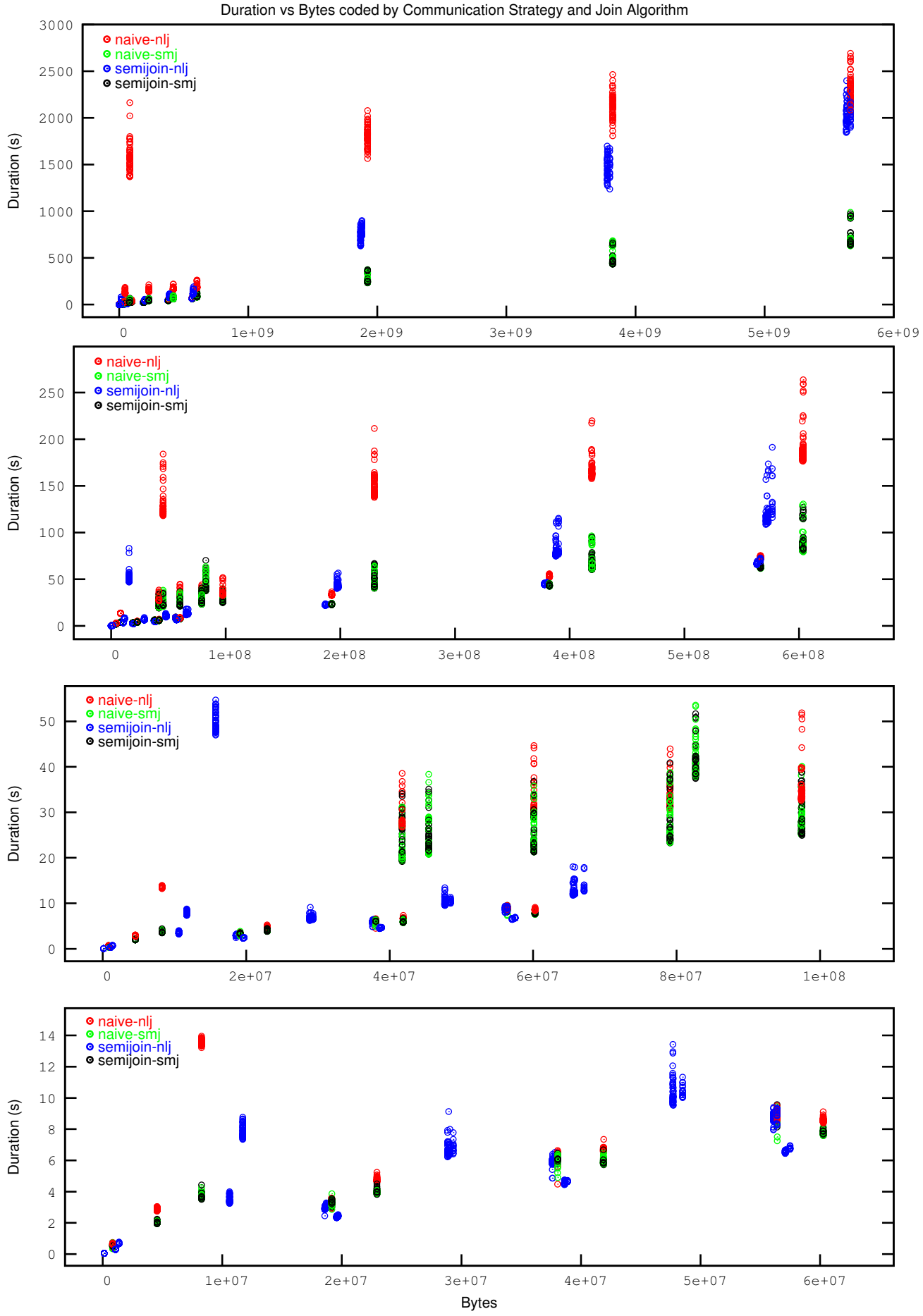
Graphs of run duration vs. number of bytes sent

The subsequent pages of this appendix include graphs in which the run duration is offset by the number of bytes sent. Each page contains four graphs. The topmost graph contains a complete overview of the experiment runs. The graphs below that are magnifications of the lower left corner of the graph above that.

Note these graphs only differ by colour coding, i.e. each of them contains the same data points. The first page is colour coded by the combined size of the input relations. Subsequently, the second is colour coded based on the combination of the communication strategy and the join algorithm used in the run. Finally, the colours in the set of graphs on the third page represent the different success ratios of the runs.


Duration vs Bytes coded by relation size
Duration (s)
Bytes
1-1
10-1
100-1
10-10
100-10
100-100
0
500
1000
1500
2000
2500
3000
1e+09
2e+09
3e+09
4e+09
5e+09
6e+09
50
100
150
200
250
1e+08
2e+08
3e+08
4e+08
5e+08
6e+08
10
20
30
40
2e+07
4e+07
6e+07
8e+07
1e+08
2
4
6
8
12
14
1e+07
2e+07
3e+07
4e+07
5e+07
6e+07


Duration vs Bytes coded by Success Ratio
SR-0
SR-33
SR-67
SR-100
Duration (s)
Bytes
0
500
1000
1500
2000
2500
3000
1e+09
2e+09
3e+09
4e+09
5e+09
6e+09
50
100
150
200
250
1e+08
2e+08
3e+08
4e+08
5e+08
6e+08
10
20
30
40
2e+07
4e+07
6e+07
8e+07
1e+08
2
4
6
8
12
14
1e+07
2e+07
3e+07
4e+07
5e+07
6e+07


Duration vs Bytes coded by Communication Strategy and Join Algorithm
naive-nlj
naive-smj
semijoin-nlj
semijoin-smj
Duration (s)
Bytes
0
500
1000
1500
2000
2500
3000
1e+09
2e+09
3e+09
4e+09
5e+09
6e+09
50
100
150
200
250
1e+08
2e+08
3e+08
4e+08
5e+08
6e+08
10
20
30
40
2e+07
4e+07
6e+07
8e+07
1e+08
2
4
6
8
12
14
1e+07
2e+07
3e+07
4e+07
5e+07
6e+07

References

[1] Berler, M., Eastman, J., Jordan, D., Russell, C., Schadow, O., Stanienda, T., and Velez, F. *The object data standard: ODMG 3.0.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[2] Bernstein, P. A., and Chiu, D.-M. W. Using semi-joins to solve relational queries. *Journal of The ACM 28* (1981), 25–40.

[3] Blasgen, M. W., and Eswaran, K. P. Storage and access in relational data bases. *IBM Systems Journal 16*, 4 (1977), 363 –377.

[4] Busse, R., Carey, M., Florescu, D., Kersten, M., Manolescu, I., Schmidt, A., and Waas, F. Xmark – an xml benchmark project, 2009. [Online; accessed 19-July-2011].

[5] Carey, M. J., Haas, L. M., Schwarz, P. M., Arya, M., Cody, W. F., Fagin, R., Flickner, M., Luniewski, A. W., Niblack, W., Petkovic, D., Thomas, J., Williams, J. H., and Wimmers, E. L. Towards heterogeneous multimedia information systems: the garlic approach. In *RIDE '95: Proceedings of the 5th International Workshop on Research Issues in Data Engineering-Distributed Object Management (RIDE-DOM'95)* (Washington, DC, USA, 1995), IEEE Computer Society, pp. 124+.

[6] Codd, E. F. *The relational model for database management: version 2.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[7] Colby, L. S. A recursive algebra for nested relations. *Information Systems 15*, 5 (1990), 567–582.

[8] Collet, C., and Vu, T.-T. Qbf: A query broker framework for adaptable query evaluationthis work is part of the mediagrid project, supported by the french aci grid program. In *Flexible Query Answering Systems.* 2004, pp. 362–375.

[9] Dogac, A., Halici, U., Kilic, E., Ozhan, G., Ozcan, F., Nural, S., Dengi, C., Mancuhan, S., Arpinar, B., Koksal, P., and Evrendilek, C. Metu interoperable database system. *SIGMOD Rec. 25*, 2 (1996), 552+.

[10] Garani, G., and Johnson, R. Joining nested relations and subrelations. *Information Systems 25*, 4 (June 2000), 287–307.

[11] Hidders, J., Kwasnikowska, N., Sroka, J., Tyszkiewicz, J., and Vandenbussche, J. Dfl: A dataflow language based on petri nets and nested relational calculus. *Information Systems 33*, 3 (May 2008), 261–284.

[12] Jaeschke, G., and Schek, H. J. Remarks on the algebra of non first normal form relations. In *PODS '82: Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems* (New York, NY, USA, 1982), ACM, pp. 124–138.

[13] Liu, H.-C., and Chirathamjaree, C. An efficient join for nested relational databases. In *Database and Expert Systems Applications*, R. Wagner and H. Thoma, Eds., vol. 1134 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, Berlin/Heidelberg, 1996, ch. 25, pp. 289–301.

[14] Liu, H. C., and Yu, J. X. Algebraic equivalences of nested relational operators. *Inf. Syst. 30*, 3 (2005), 167–204.

[15] Paredaens, J., and Van Gucht, D. Converting nested algebra expressions into flat algebra expressions. *ACM Trans. Database Syst. 17*, 1 (1992), 65–93.

[16] Roth, M. A., Korth, H. F., and Silberschatz, A. Extended algebra and calculus for nested relational databases. *ACM Trans. Database Syst. 13*, 4 (1988), 389–417.

[17] Silberschatz, A., Korth, H., and Sudarshan, S. *Database Systems Concepts*. McGraw-Hill, Inc., New York, NY, USA, 2006.

[18] Tomasic, A., Raschid, L., and Valduriez, P. Scaling access to heterogeneous data sources with disco. *IEEE Transactions on Knowledge and Data Engineering 10*, 5 (Sept 1998), 808–823.

[19] Wiederhold, G. Mediators in the architecture of future information systems. *IEEE Computer 25*, 3 (March 1992), 38–49.

[20] Wiederhold, G. Glossary: Intelligent integration of information. *Journal of Intelligent Information Systems 6*, 2 (June 1996), 281–291.