Complex Object Joins in a Distributed Database

Wouter B. Teeuw* Henk M. Blanken

University of Twente, Department of Computer Science, Information Systems — Databases
P.O. Box 217, NL–7500 AE Enschede, The Netherlands, E-mail: {teeuw,blanken}@cs.utwente.nl

Abstract

In non-standard database systems performance tends to be critical. To obtain the required response times for processing complex objects, parallelism needs to be exploited and suitable storage structure for complex objects have to be used. In this paper we use an hierarchical complex object model with object references. The join operations for these complex objects, called tuple-objects, are categorized into materialized, functional and value-based joins. The performance of these joins is analytically evaluated, giving insights into which distributed storage structures for tuple-objects are most effective for the different kinds of joins.

Keywords: Complex objects, Distributed databases, Join operation, Performance aspects, Storage structures.

1 Introduction

Data-intensive database applications such as robotics, cartography and CAD/CAM, the so-called *non-standard* database applications, require a high performance in retrieving and processing *complex objects*. A complex object is a large cluster of structured data that forms a logical unit. In general, it is built by applying various complex object constructors to other complex objects or basic values. The constructors that the system should at least support seem to be tuple, set and list. Also, a complex object has some notion of object identity and complex objects have relationships to each other. Those complex objects need to be stored efficiently in order to achieve the required responsiveness.

Without special provisions the relational storage model is not very well suited to manage complex objects. Since a complex object will be represented by several tuples of several relations, a large number of *join* operations will be required to reassemble the complex object from the corresponding database relations. Those time and resource consuming joins cannot be accepted if performance is critical. Therefore alternative storage models for complex objects have been proposed. Extensions of the relational model may keep the advantages of relational storage while at the same time providing a better performance. For example, complex object may be represented in a nested relational data model [8,10]. Nested relations might be stored in a traditional relational database. To preserve the nested structure, join indices [14] or tuple identifiers [3,7] can be used. Alternatively, the nested tuples might be stored contiguously on disk [5,9]. A completely non-relational modeling of complex objects is provided by the molecule-atom data model (MAD model), in which complex objects are modeled as a set of atoms linked together in a network-like structure [6].

In this paper we analyse the performance aspects of several storage structures for complex objects. In most earlier work only object retrieval is considered and no attention is paid to

*The investigations were partly supported by the Foundation for Computer Science in the Netherlands SION under project 612-317-025 nicknamed Starfish.

a further processing of the objects. In particular, since complex objects tend to be retrieved in their entirety, the selection of entire objects is considered [12]. We investigate the effect of join queries on the retrieval of complex objects. Since we need a model for what a complex object is, we introduce the notions of *tuple-objects* and *composed objects* (section 2). A tuple-object is a kind of nested tuple. Its attributes are existence dependent. Tuple-objects are the unit of sharing among a number of composed objects. Notice that we do not introduce yet another complex object model. Rather, we provide a view on complex objects which enables a performance evaluation.

Similar to the relational join, a tuple-object join might restrict the instances of the one tuple-object type based on the attribute values of another tuple-object type. However, in object-oriented databases the traditional value-based join seems to play a less dominant role. More important is the object traversal along references, leading from one tuple-object instance to another. This navigation in a composed object is called a join as well (implicit or functional join). Moreover, a relation valued attribute might materialize what used to be a join in a traditional relational system. A further description of tuple-object joins is presented in section 3. Next, in section 4, we evaluate which storage structures are most effective with regard to these join queries. Three storage structures get attention, two of which are normalized with special features. Performance will be measured in terms of response times. The evaluation will be analytical, with disk I/O, network message, and processor (CPU) load being considered separately. Results are presented in section 5.

We focus on a distributed shared-nothing database system based on a local area network. Distributed systems are more and more used because they increase reliability, availability and in particular performance. Moreover, shared-nothing systems have demonstrated speedup and scale-up to hundreds of nodes and seem to be the basis for distributed database technology [4]. Our performance evaluation that is based on join queries shows the costs of different complex object storage structures given various parameter values. Therefore, together with identical results for selection queries, it may be the base for a query optimizer and/or a data allocation manager for complex objects in a distributed database system.

2 Complex objects

2.1 Tuple-objects and composed objects

In this section we describe our vision on a complex object, which we call a *tuple-object*. A tuple-object is constructed from some basic types (such as boolean, character, integer, real, string) by applying the tuple, set and list constructors. Among the basic types are an identifier type and a reference type as well. The constructors are completely orthogonal, i.e. they can be applied in any order. However, the top level construct must be a tuple. Tuple-objects are instances of a tuple-object *type*. All tuple-objects of the same type have an identical structure. Connected to each tuple-object is a unique system generated *identifier* or surrogate of the type OID. It distinguishes the object from all others. Identifiers are unique, can't be affected by user updates and are used by tuple-objects to *refer to each other*.

Figure 1 shows an example of a tuple-object of the type Doctor. The symbols $\langle\ \rangle$ denote a tuple, $\{\ \}$ a set and [ ] a list. A tuple-object of type Doctor is a tuple containing four attributes: two atomic attributes (identifier and name) and two non-atomic attributes. The non-atomic attribute private is a tuple containing two attributes that are both non-atomic again: address is a tuple of three atomic attributes and phone is a list of atomic values. The non-atomic attribute specialization is a set of atomic values. The type REF(Disease), a reference type, has as potential values the identifiers of the tuple-objects of the type called Disease.

A tuple-object is hierarchically structured and resembles the NF^2 and extended NF^2 data

```
TUPLE-OBJECT Doctor = {⟨
          identifier:       OID,
          name:             STRING,
          private:          ⟨  address:  ⟨  street:  STRING,
                                            nr:      INTEGER,
                                            city:    STRING  ⟩,
                               phone:    [INTEGER]  ⟩,
          specialization:   {REF(Disease)}
⟩}
```

Figure 1: Declaration of a Doctor tuple-object.

models [8,10]. The characteristic property of a tuple-object is its *existence dependency.* The removal of a tuple-object includes the deletion of all its components. As a consequence there is no sharing of data between tuple-objects. However, data sharing is still possible since a tuple-object as a whole is the *unit of sharing* among a number of composed objects. A *composed object* is a collection of tuple-objects linked together by the references between them. Starting from a specific *root* tuple-object, the whole composed object can be retrieved by traversing these links. Again, we may consider types and instances. Composed objects resemble the molecules in the MAD model [6].

2.2 A direct storage model for tuple-objects (DSM)

Tuple-objects may be mapped directly into a single storage unit and, as far as possible, be stored contiguously on disk. Storing tuple-objects (or rather complex objects) as a whole is called *direct storage model* [15], which we will refer to as *DSM.* In the direct storage model there is a 1–1 correspondence between the conceptual tuple-object and the internally stored object. Obviously the retrieval of an entire tuple-object will be efficient. The retrieval of a few attributes, on the contrary, may be inefficient since probably the whole tuple-object has to be retrieved. Figure 2 shows the direct representation of the Doctor tuple-object of figure 1. The figures 3 through 5 show yet three other example tuple-objects that we will use throughout this paper. Tuple-object occurrences of the same type form a single nested relation. We will refer to these relations as the DOCTOR, PATIENT, ILLNESS and HOSPITAL relations respectively.

Several implementations for DSM exist [5,9]. In our performance evaluation we assume an arbitrary implementation, provided that all tuple-objects of a single type are stored, one behind another, in a single sequence of pages (file). The effective page occupation will be less than 100% due to the fact that storage space may be wasted or used for structure information. We assume an *index on the tuple-object identifier*, which translates this identifier into the necessary page information. In our distributed shared-nothing environment the homogeneous nodes are connected by a local area network. Each node consists of a processor, internal memory and disk drive. The nodes communicate to each other by message passing over the network, which is the

DOCTOR

identifier$_{oid}$	name$_{string}$	⟨private⟩				{specialization$_{ref(\text{Illness})}$}
		⟨address⟩			[phone$_{integer}$]	
		street$_{string}$	nr$_{integer}$	city$_{string}$		

Figure 2: Direct representation of the Doctor tuple-object

PATIENT

identifier$_{oid}$	name$_{string}$	⟨private⟩					{⟨illness⟩}		
		⟨address⟩			job$_{string}$	phone$_{integer}$	disease$_{ref(\text{Illness})}$	from$_{real}$	till$_{real}$
		street$_{string}$	nr$_{integer}$	city$_{string}$					

Figure 3: Direct representation of the Patient tuple-object

ILLNESS

identifier$_{oid}$	name$_{string}$	{⟨symptom⟩}		treatment$_{string}$
		sname$_{string}$	description$_{string}$	

Figure 4: Direct representation of the Illness tuple-object

only resource they share. Each node owns a disjunct portion of the database data. So with DSM the tuple-objects will be randomly (and equally) distributed over the nodes. Each node stores a horizontal fragment of the nested relation that is formed by all tuple-objects of the same type.

2.3 A normalized storage model for tuple-objects (NSM)

The hierarchically structured tuple-object can be seen as a nested tuple, a tuple with relation valued attributes. However, since set and list attributes do not necessarily consist of tuples, we have to regard each set as a set of tuples (= a relation) and each list as a list of tuples (= an ordered relation). For example, a set of integers is considered as a set of unary tuples with an integer valued attribute.We may store all the sub-relations of a nested relation independently in traditional flat relations. We refer to this storage model as *normalized storage model* or *NSM.*

NSM provides a better performance for partial tuple-object retrieval. It allows to retrieve first those parts of the tuple-object that have the highest probability to make further disk I/Os superfluous. Projection on many attributes is one of the best supported operations. In general, the retrieval of an entire tuple-object is acceptably efficient if additional support for the reconstruction of the tuple-object is provided. Such additional support generally consists of a mechanism that appends to each tuple in each relation a unique identifier. These identifiers are used to store the tuple-object structure, either by using them as pointer values [1,14], or by constructing them in such a way that they contain information about root and parent tuples [3, 7].

We assume an NSM implementation in which each tuple in each relation has a tuple-identifier (tid) consisting of three parts. The first part is the identifier of the tuple-object the tuple belongs to ('root part'). The second part is an identification of the parent non-atomic attribute of this tuple ('parent part'). Finally, there is an identifier for the tuple itself ('own part'). In this way, the hierarchical tuple-object structure is tied up in the tid's and the normalization of a nested tuple and its inverse are unambiguous. Notice that a tid is generated by the system and invisible to the users. The tuples of each relation with an equal root part in the tid are clustered together. Within such a cluster, the same holds for tuples with an equal parent part of the tid. In this way the tids not only keep the tuple-object structure, but also enhance the performance

HOSPITAL

identifier$_{oid}$	name$_{string}$	city$_{string}$	{doctor$_{ref(\text{Doctor})}$}	{patient$_{ref(\text{Patient})}$}

Figure 5: Direct representation of the Hospital tuple-object

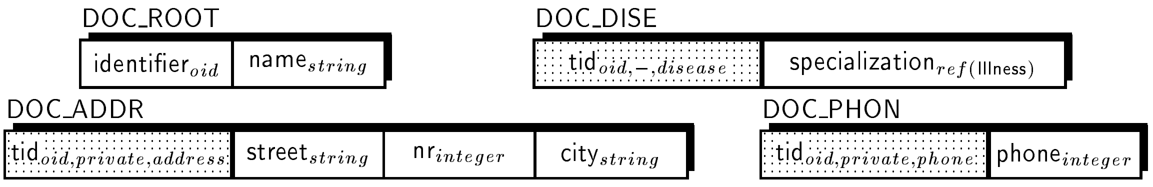

Figure 6: Normalized representation of the Doctor tuple-object

of tuple-object retrieval. We assume that each flat relation is stored in a separate file (sequence of pages). The pages of these file contain some wasted space. An index on the root part of the tid exists. This index is not necessarily dense. Per relation, one entry per tuple-object instance or per page may be sufficient.

Figure 6 shows a normalized representation of the Doctor tuple-object. Tuple-objects of the type Doctor are stored in four flat relations, referred to as DOC_ROOT, DOC_ADDR, DOC_PHON and DOC_DISE. Notice that there is no 'DOC_PRIV' relation because the private attribute is a tuple that contains only non-atomic attributes. The identifier attribute in the DOC_ROOT relation makes a tid superfluous. Since a list of integers is regarded to as a list of unary tuples, the own part of the tid in the DOC_PHON relation is related to the phone attribute and may be used to preserve the list ordering. The tid's in the DOC_DISE relation have no parent part since on the first level of nesting the parent is equal to the root.

In a distributed environment, the tuples of all relations are distributed over the nodes. We may distinguish two situations. First, all flat tuples belonging to a single tuple-object occurrence may be allocated to a single node. We will refer to this case as *t-NSM* (tuple-objects on a node NSM). Second, the tuples of a single tuple-object instance may be randomly (and per relation equally) distributed over the nodes. We refer to this case as *f-NSM* (fragments of tuple-objects on a node NSM).

3 Joins on tuple-objects

In a traditional relational environment the join of two relations applies a selection on the Cartesian product of these two relations. Actually, the tuples of the one relation are selected based on the attribute values in the other relation (primary/foreign keys). In database systems with objects and object references this join based on matching attribute value pairs plays a less dominant role. More important are object accesses along reference chains leading from one object instance to another. This object traversal is called a join as well, the so-called implicit or functional joins. In general, a join for tuple-objects is an operation that correlates different (complex) attributes of arbitrary tuple-object types. We will make a distinction between three different kinds of joins: materialized, functional and value-based joins. In all cases, we consider the results of the join queries as data (structured values) rather than tuple-objects. Using the tuple-object types as described in section 2, we will show some examples. An extended version of SQL is used to express the example queries. In section 4 the example queries will be used to evaluate the storage models for tuple-objects in a distributed database.

3.1 Materialized joins

Since tuple-object attributes may be complex valued, operations that would be a join in the traditional relational environment might become the retrieval of some attributes with tuple-objects. The join has been materialized in the tuple-object structure.

Example 1 For each illness we want to know all symptoms. The output is a set of binary tuples. The first attribute is the name of the illness, the second attribute is a set of binary tuples, the attributes giving the symptom name and description respectively.

```
SELECT   {⟨name, symptom⟩}
FROM     Illness
```

The brackets in the SELECT clause show that in this case the result is a set of binary tuples. Notice that the non-atomic attribute symptom in the Illness tuple-object is a set of (binary) tuples. □

3.2 Functional joins

Tuple-objects are connected to each other by links. The navigation from tuple-object to tuple-object by following these links means joining them (implicit joins).

Example 2 We want to find all doctors who are working in the hospital called 'central'. The output contains all the available data of these doctors.

```
SELECT   {d}
FROM     d IN Doctor,
         h IN Hospital
WHERE    h.name = 'central' AND
         d.id IN h.doctor
```

The brackets in the SELECT clause show that in this case the result is a set. Since d is a tuple-object, the result is a set of (nested) tuples, each tuple containing the data of a doctor. □

The example query consists of two parts. First the selection of (a) tuple-object(s) of the type Hospital where the value of the attribute name is 'central'. Second the selection of a number of tuple-objects of the type Doctor. The identifiers of these tuple-objects have to be in the set attribute doctor of a selected Hospital tuple-object. This second part of the query involves the traversal of links between tuple-objects. It is the selection of a number of tuple-objects of a certain type where the identifiers are given by the value of an attribute of another tuple-object instance. This latter part of the query is what we call a functional join. A functional join is always an equi-join since identifiers can't be ordered. Also, a functional join between two tuple-objects does not involve multiple join attributes.

3.3 Value-based joins

Tuple-objects can be joined in the traditional relational sense: the Cartesian product of two sets of tuple-objects and a selection on the result. The selection predicate tests the attribute values of the tuple-objects. However, there are some important differences with the traditional relational join. The atomic join attributes in a single tuple-object are not necessary part of the same 'sub-relation' of the hierarchically structured tuple-object. In section 4 we will see that whether the join attributes in a (single) tuple-object are part of the same sub-relation or not influences the performance of the join query. Such problems do not occur in the first normal form relational model.

Also, the join attributes may be tuple, set or list constructed rather than atomic. As a consequence, the join predicate might contain operators specific for sets, like SUBSET ($\subset$), SUPERSET ($\supset$) or IN ($\in$)[1], or lists, like HEAD and TAIL. With non-atomic join attributes, tuples, sets or lists have to be compared with identically structured values. Since tuples, sets and lists may contain non-atomic attributes again, this process is a recursive one. Notice that only identically structured attributes can be compared with each other. For example, we can not compare a set

[1] Actually we already used the IN operator in example 2, though that was just a simple non-recursive case

of integers with a set of characters and even not with a set of unary tuples containing an integer. The next example shows the use of set inclusion in the join predicate.

Example 3 We want to find the names of the doctors who have been specialized in all the diseases Mr. Brown suffers from. The output is a set of doctor names.

```
SELECT   {d.name}
FROM     d IN Doctor,
         p IN Patient
WHERE    d.specialization ⊇ ( SELECT   {i.disease}
                              FROM     i IN p.illness
                              WHERE    p.name = 'Brown' )
```
□

The latter example query appears to be what we call a DIVIDE operation in the traditional relational algebra. Set inclusion showing up in the join predicate is unknown to the traditional relational model. However, though equality tests may be more complex and performance aspects may be different, the general idea of comparing sets is not different from comparing atomic values.

4 Performance evaluation

4.1 Building the query results from the base (nested) relations

The SQL-like queries as presented in the previous section are translated to some kind of relational algebra which has been generalized for tuple-objects. The resulting equations show how the join results are built from the basic database relations. These basic relations may be distributed or nested. The translation into algebraic equations has been performed manually, using some heuristics. It is supposed to be a very clever (or even optimal) translation. The next subsection shows how to translate these equations into system response times. For reasons of space, only a global indication of our cost analysis is given. A more detailed presentation can be found in an additional report [13].

While constructing algebraic equations for the example queries, we used some basic assumptions about the query execution strategies. Since we may not expect from a user to be able to handle large tuple-objects that have been represented as a first normal form relation, results might be represented in a direct (DSM) form, even if NSM is used. This is no problem since results are not generated to be stored in the database, but to be shown to a user. Also, message passing over the network is minimized. CPU time might even be offered in order to reducing the network load. Therefore, as much as possible the data are processed where they have been allocated. Parallelism is used as much as possible. The results are always sent over the network to an output, which is not one of the N nodes in the system. Finally, all relations are equally distributed over the nodes. Of course, this is true only as far as the number of tuples and the clustering mechanisms (NSM) allow such a distribution. With DSM and t-NSM it is known which tuple-objects have been stored on which nodes. With f-NSM it is known on which nodes the root tuples of the tuple-objects have been stored. We assume that a hash function applied on the tuple-object identifier gives this node information.

We show a few examples. With DSM the query of example 2, find the doctors working in the hospital 'central', becomes (the notation should be rather self explanatory):

$$\mathsf{TEMPI}_\imath := (\sigma_{name=\text{`central'}}\mathsf{HOSPITAL}_\imath)\cdot\mathsf{doctor}\quad (1 \le \imath \le \mathrm{N}),$$

$$\mathsf{RESULT2} := \bigcup_{\imath=1}^{\mathrm{N}}(\mathsf{TEMPI}_\jmath \rtimes_{doctor=identifier}\mathsf{DOCTOR}_\imath)\quad (\mathsf{TEMPI}_\jmath \neq \emptyset) \tag{1}$$

In parallel all N nodes perform a selection on the HOSPITAL nested relation fragment they own, and project matching tuples on the doctor attribute. As we assume that there is only a single hospital called 'central', all $\text{TEMPI}_\imath$ are empty except for $\text{TEMPI}_\jmath$, which is a set of doctor identifiers and has to be replicated on all nodes. Replicating a relation (or fragment) on all nodes means creating a copy of the entire relation (fragment) on all nodes. However, since it is known which tuple-objects are stored on which nodes, $\text{TEMPI}_\jmath$ will be distributed (rather than replicated) over the nodes. That is, each tuple of $\text{TEMPI}_\jmath$ is only sent to a single node only. $\text{TEMPI}_\jmath$ is used by all nodes to select the corresponding doctors (the semi-join), whereupon the result is collected (the union).

The next example shows how the result of the query of example 1 (names and symptoms of all illnesses) will be built with f-NSM. ILL_ROOT is the vertical fragment of ILLNESS containing the attributes identifier, name and treatment. ILL_SYMP contains sname and description.

$$\mathsf{RESULT1} := \bigcup_{\jmath=1}^{N}(\pi_{name,sname,description}(\mathsf{ILL\_ROOT}_\jmath \bowtie_{identifier=tid^{(r)}} (\bigcup_{\imath=1}^{N} \mathsf{ILL\_SYMP}_\imath))) \qquad (2)$$

First, ILL_SYMP is replicated on all nodes as shown by the second union symbol. Actually ILL_SYMP need not to be replicated entirely on all nodes. Rather, ILL_SYMP will be (re)distributed over the nodes since for each tuple in ILL_SYMP there is only a single corresponding tuple in ILL_ROOT. The node this particular tuple has been stored on can be found by applying a hash function on the tuple-object identifier that has been stored in the tid attribute. Notice that by using intra tuple-object clustering (t-NSM) this redistribution step could be omitted. In parallel the nodes construct result tuples by executing a join and a projection. Finally the result is sent to the output (the union). The result is a traditional relation.

4.2 How to compute the response times

The algebraic formulas have to be translated into average response times. Using these times, the performance of the different storage structures under different system loads (join queries) can be compared. Given the equations, the queries will be evaluated as efficiently as possible. Selections, projections and dot operations are combined to a single action in order to save CPU and IO time. To prevent intermediate results from being stored on disk, pipelining is used wherever possible. In particular, when relations are redistributed over nodes the received tuples are processed in a pipelined way. Pipelining is used even when a join (to reconstruct doctors) and a division (to construct result) are executed behind each other (example 3, NSM).

We need some parameter settings to be able to determine response times. Table 1 shows the system system parameters. The parameters should be self-explanatory. Most settings have been based on values as presented in the literature [11]. We need some tuple-object parameters as well. We assume the next *average* values. A tuple-object of the type Doctor contains 2 phone numbers and 100 references to a specialization. A Patient tuple-object has 5 illnesses. Each Illness tuple-object has 5 symptoms. A tuple-object of type Hospital has 25 references to a doctor and 500 references to a patient. Finally, a string attribute is 40 bytes and all other attributes (inclusive identifier attributes) are 8 bytes.

We present the I/O, network and CPU times independently of each other. The advantage of such an approach is that it facilitates the conversion (adaptation) of our conclusions to other systems or new technologies. The following notation is used. If R is a (nested) relation, $||\mathsf{R}||$ is the number of tuples in R, *tuple_size*(R) is the size of a tuple in R and $|\mathsf{R}|$ is the number of pages used to store R on. Obviously $|\mathsf{R}| = ||\mathsf{R}|| * \textit{tuple\_size}(\mathsf{R}) * \mathrm{F}_{pgoc}/\mathrm{PG}$. Notice that the CPU costs due to disk I/O and message passing are not included in the processor costs, but in the disk I/O and message costs respectively. Therefore the disk I/O, message and processor times may be added to a single response time, not considering a possible overlap.

Parameter	*Description*	*Setting*	
F_{hash}	comparison overhead factor for a hash join	1.2	
F_{pgoc}	number of pages to store PG byte of data (= 100% / degree of page occupation)	1.4	($\simeq$ 100%/70%)
M	main memory size per node for data	512	pages (= 1 Mbyte)[2]
N	number of nodes in system	10	nodes
PG	size of a memory/disk page	2048	byte
sel	query selectivity (divide operation)	0.01	
t_{comp}	CPU time to compare two values	0.2	μs
t_{hash}	CPU time to hash a value	0.9	μs
t_{move}	CPU time to move data	0.1	μs/byte
t_{IO}	time to read/write a disk page	20	ms/page
t_{MSG}	average time to send a network message	1.5	μs/byte[3]

Table 1: Default system parameters.

Disk I/O

On each node local indices map tuple-object identifiers to page numbers. With our default values, index accesses appear to be no bottleneck. Therefore, we assume a tuple-object identifier can be transformed to a physical page address without any costs to be considered. Each page access takes t_{IO} ms. All nodes may input data in parallel. So if a (nested) relation R has been distributed over N nodes, the time needed to retrieve relation R from disk will be $(|\mathsf{R}|/\mathrm{N}) * t_{IO}$.

If only some of the tuple(-object)s that have been stored on a sequence of pp pages are needed, possibly not all the pp pages need to be fetched from disk. Suppose tf randomly distributed tuple(-object)s have to be fetched, given their identifiers. Bernstein [2] derived a formula for the number of pages to input:

$$\mathit{page\_fetches}(\mathsf{pp},\mathsf{tf}) = \begin{array}{lll} \mathsf{tf} & \mathit{if} & \mathsf{tf} \leq \mathsf{pp}/2 \\ (\mathsf{tf}+\mathsf{pp})/3 & \mathit{if} & \mathsf{pp}/2 < \mathsf{tf} \leq 2 * \mathsf{pp} \\ \mathsf{pp} & \mathit{if} & 2 * \mathsf{pp} < \mathsf{tf} \end{array} \tag{3}$$

The formula assumes that tuples do not span pages. However, with large tuples or even entire tuple-objects, the tuple(-object) size ts may be larger than the disk page size PG. Therefore we use the next formula as an estimate for the number of page accesses.

$$\mathit{pages}(\mathsf{pp},\mathsf{tf},\mathsf{ts}) = \mathsf{tf} * \lfloor \frac{\mathsf{ts} - 0.5}{\mathsf{PG}} \rfloor + \mathit{page\_fetches}(\mathsf{pp}\text{-}\mathsf{tf} * \lfloor \tfrac{\mathsf{ts}-0.5}{\mathsf{PG}} \rfloor, \mathsf{tf}) \tag{4}$$

The formula can be explained as follows. If a tuple(-object) is smaller in size than a page, it is just Bernstein's formula. Since the unit of size is an *entire* byte, the '0.5' takes care of rounding problems. If a tuple(-object) is larger in size than a page, preferably it will be stored on a minimum number of pages. Since a tuple(-object) is stored contiguously on disk, $\lfloor \frac{\mathsf{ts}-0.5}{\mathsf{PG}} \rfloor$ page fetches per tuple(-object) fetch are necessary anyhow. For the tf tuple-object parts that still need to be fetched from the remaining pages we use Bernstein's formula.

The tuples might be clustered, rather than randomly distributed over the pages. Suppose tf tuple(-object)s have to be fetched. They have been clustered in groups of cs tuple(-object)s. Entire clusters are fetched from disk. The size of a tuple(-object) is ts and the clusters have been randomly distributed over the pp pages. Then, the number of page accesses $\mathit{pages}^c(\mathsf{pp},\mathsf{tf},\mathsf{ts},\mathsf{cs})$ becomes:

$$\mathit{pages}^c(\mathsf{pp},\mathsf{tf},\mathsf{ts},\mathsf{cs}) = \mathit{pages}(\mathsf{pp}, \tfrac{\mathsf{tf}}{\mathsf{cs}}, \mathsf{ts} * \mathsf{cs}) \tag{5}$$

[2]This value seems ridiculously small. However, our example tuple-objects are small as well (in size and number).

[3]Obviously t_{MSG} might depend on the message size since there will be a fixed overhead per message that is independent of the message size. However, we assumed an average message transfer rate in which message set up times have been included.

Network messages

The communication is on a point-to-point basis (no broadcasting). So, as opposed to disk I/O and processor times, communication times are added if two or more nodes send in parallel. We do not consider collisions explicitly, but assume that the time for retransmitting messages has been included in the network parameter t_{MSG}. If a (nested) relation R has to be sent over the network the corresponding network message time is $||R||*\textit{tuple\_size}(R)*t_{MSG}$. Often occurring situations are that a relation fragment is either replicated on all nodes in its entirety, or fragmented and distributed over the nodes.

Processor (CPU) time

We illustrate the processor costs by examining the join $R \bowtie S$. All other operations are handled in a rather analogous way. Suppose a hash-based join algorithm with R as the inner (smaller) relation. First a hash table is built for the inner relation. The processor costs: $||R||*(t_{hash}+\textit{tuple\_size}(R)*t_{move})$. Then the outer relation is probed against this hash table: $||S||*(t_{hash}+F_{hash}*t_{comp})$. Finally a result tuple is constructed for each pair of matching join attributes: $||R||*||S||*\text{sel}*(\textit{tuple\_size}(R)+\textit{tuple\_size}(S))*t_{move}$. The parameter 'sel' is used to indicate the join selectivity. If the inner relation does not fit in main memory, both operand relations will be partitioned into a number of buckets by hashing on the join attribute. In this way a large join is split into a number of smaller joins for which enough main memory space is available. The buckets will be temporarily stored on disk. The total costs for a bucket forming phase will be $||R||*(t_{hash}+\textit{tuple\_size}(R)*t_{move}) + ||S||*(t_{hash}+\textit{tuple\_size}(S)*t_{move}) + (|R|+|S|)*2*t_{IO}$. If the join attribute is the identifier attribute, on which an index exists, a bucket forming phase can be omitted. For, since there is clustering on identifiers a partition based on the join attribute already exists. With a semi-join $R \rtimes S$ the result is constructed from the attributes of S only. Consequently $\textit{tuple\_size}(R)+\textit{tuple\_size}(S)$ becomes $\textit{tuple\_size}(S)$ in the result construction phase.

5 Results

5.1 Response times for example 1 (materialized join)

Table 2 shows some response times if our default parameter values are used. We analyze the results query by query. For the query of example 1, the selection of all Illness names and symptoms, the disk I/O time for all three storage structures is about equal (figure 7). All data of all tuple-objects of the type Illness has to be retrieved from disk. With NSM the Illness data

	DISK I/O	+	MESSAGES	+	PROCESSOR	=	TOTAL
	example 1 (a materialized join)						
DSM	6.67	+	6.60	+	0.0440	=	13.3
f-NSM	7.22	+	14.9	+	0.172	=	22.3
t-NSM	7.22	+	9.00	+	0.0754	=	16.3
	example 2 (a functional join)						
DSM	0.636	+	0.0360	+	0.000311	=	0.673
f-NSM	0.741	+	5.87	+	0.0993	=	6.71
t-NSM	0.232	+	0.0360	+	0.0303	=	0.298
	example 3 (a value-based join)						
DSM	24.0	+	0.00204	+	0.0605	=	24.1
f-NSM	8.93	+	5.40	+	0.252	=	14.6
t-NSM	8.93	+	0.00204	+	0.189	=	9.13

Table 2: Calculated default response times (s) for the example join queries.

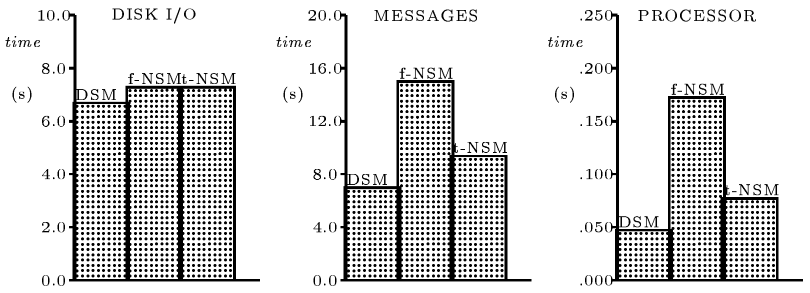

Figure 7: I/O, message and CPU times for the query of example 1 ('materialized' join).

is distributed over two relations. Therefore DSM has a small advantage in I/O time but the advantage is small, about 10%.

The network message cost are minimal for DSM as well. Only for sending the result to the output communication is needed. This holds true for t-NSM as well, but then the result is larger since it is in a relational (rather than direct) representation. With f-NSM, much communication is needed since initially the illness' name and symptoms are not necessarily stored on the same node. N.B. The vertical scales in figure 7 are different!

As with the message costs, the processor costs are smallest for DSM and largest for f-NSM. In the former case there is only a projection on the right attributes. In the latter case there is a real join (due to the normalized storage model). Moreover, since there is no intra tuple-object clustering with f-NSM, processor time is involved with redistributing the relations in order to get the data that has to be joined on the same node. Notice that the CPU times as presented in this paper only include the CPU time as relevant for the specific join algorithms. Not included are the CPU times involved with disk I/O and message passing, which have been included in the correspondingly named times. The CPU time for swapping, garbage collection, etc. has not been considered too. Therefore the actual processor load will be much larger.

With other parameter choices the trends remain identical, except for one particular point to mention. The response time for DSM is (for this example query) rather independent of the main memory size. However, with NSM we have joins. If the main memory is too small (or the local relations too large) the large join has to be split into a number of smaller joins. Such a bucket forming phase, which was not needed with the default values, involves much processor and disk I/O time. Moreover, with f-NSM a small main memory will cause data that is received from the other nodes after redistribution to be temporarily stored on disk. This effect can be seen from

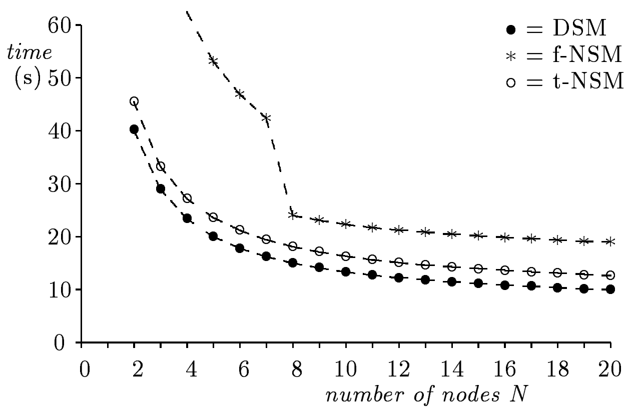

Figure 8: Total response time while varying the number of nodes with the query of example 1

figure 8. With less than eight nodes the relation ILL_SYMP can not be kept in main memory after its redistribution and the response time for f-NSM increases. As indicated by figure 8, for all storage models the total response time (t_{IO}+t_{MSG}+t_{CPU}) decreases with an increasing number of nodes. The disk I/O and processor load is distributed over more nodes. If N $\rightarrow \infty$ the total response time even approaches the communication time t_{MSG}. The communication time is independent of N with DSM and t-NSM and slightly increases with N for f-NSM, since distributing the data over more nodes makes the probability that a tuple has initially been stored on the correct node (where it has to be after redistribution) smaller.

Concluding, for this example query the overall costs are minimal with a direct storage model. A normalized storage model is worse, but with intra tuple-object clustering it is not much worse. A normalized model, and in particular one without intra tuple-object clustering is more sensitive to the available main memory.

5.2 Response times for example 2 (functional join)

For the query of example 2, select all doctors working in the hospital 'central', the disk I/O time is minimal for t-NSM (figure 9). The I/O time of DSM is worse since DSM inputs all tuple-objects of the type Hospital in their entirety, though only a single one is needed. A normalized model, on the contrary, only inputs the root tuples in their entirety. From these root tuples the identifier of the relevant hospital is found. Only for the selected tuple-object the remaining data is retrieved. However, for the second part of the query, namely the input of entire tuple-objects of the type Doctor where the identifiers are known, DSM appears to be faster than NSM with regard to the disk I/O time. For, with DSM the data is clustered. With t-NSM all tuples belonging to a single doctor are stored on a single node, clustered together per relation. With f-NSM these relation fragments are distributed over the nodes. This makes the average cluster size of f-NSM smaller as compared with t-NSM. Therefore more and smaller clusters of tuples have to be retrieved, which makes the disk I/O less effective. Therefore, the disk I/O time of f-NSM is the worst of all.

The communication time is worst for f-NSM and almost entirely determines the total response time for this storage structure. All the data belonging to a tuple-object of the type Doctor has to be collected on a single node to make its reconstruction possible. Many network messages are involved. On the contrary, the communication costs for the other storage models is almost negligible. Since there's almost no communication with DSM and t-NSM, the total response time (t_{IO}+t_{MSG}+t_{CPU}) decreases if the number of nodes is increased (not shown in a figure). For, the disk I/O and CPU load are distributed over more nodes. On the contrary, with f-NSM the total response time is determined by t_{MSG} and increases with an increasing number of nodes. As long as there is no main memory overflow, this increase is only small.

The processor time is minimal with DSM since, as opposed to the normalized models, no

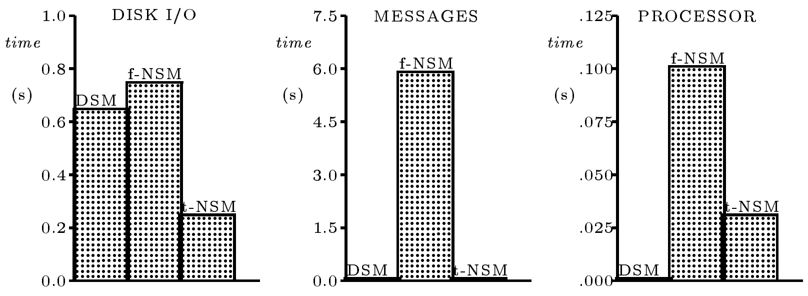

Figure 9: I/O, message and CPU times for the query of example 2 ('functional' join).

joins are needed to reconstruct a tuple-object from its normalized representation. In order to collect the information of each doctor on a single node, f-NSM has to redistribute relations over the nodes, involving a hash on each tuple, thus involving additional CPU time.

Deviating from our default values and assumptions, f-NSM needs special attention. As opposed to DSM and t-NSM, which both exploit parallel disk I/O on an inter tuple-object level, f-NSM uses intra tuple-object parallelism with regard to disk I/O (for a single tuple-object has been distributed over multiple disks). In the example query, a number of tuple-objects of the type Doctor are fetched from disk in parallel. Suppose we have either skew (the tuple-objects that have to be fetched are not equally distributed over the nodes) or only a few number of tuple-objects to be retrieved (there are more nodes than tuple-objects). In both cases f-NSM has the advantage over DSM and t-NSM to enable more parallel disk I/Os [12].

Concluding, for this example query the costs are minimal with a normalized storage model with intra tuple-object clustering. A normalized storage model without intra tuple-object clustering requires too much communication to get acceptable response times. A direct storage model, fast though it is in fetching the Doctor tuple-objects, is less flexible in selecting a single tuple-object and therefore results in a non-optimal overall performance.

5.3 Response times for example 3 (value-based join)

For the query of example 3, find the names of the doctors who can treat all Mr. Brown's diseases, the disk I/O time is minimal with the normalized storage models (figure 10). This, again, is due to the fact that DSM inputs all tuple-objects in their entirety. In particular this is ineffective with the Patient tuple-objects. Although only the attribute name has to be tested to be valued 'Mr. Brown', entire tuple-objects are retrieved. The normalized models are less bothered by retrieving superfluous information.

As with the query of example 2, f-NSM severely suffers from communication overhead. Again processor costs are minimal with DSM (no joins to reconstruct a tuple-object from its normalized representation) and maximal with f-NSM (the overhead for redistributing data over the nodes). It is interesting to notice that, once the data have been fetched from disk and shipped to the right nodes, the specific operation that implements the value-based join (namely the divide) is as expensive for all storage models. Therefore we expect the same trends in the results of other (application specific) value-based joins.

Varying the number of nodes in the system (figure 11) shows that with DSM and t-NSM the total response time is almost inversely proportional to the number of nodes. This is obvious since the response time is almost entirely determined by disk I/Os. These disk I/Os are distributed over the nodes. With f-NSM, main memory overflow occurs if the number of nodes becomes smaller than six. Then the relation DOC_DISE will be temporarily stored on disk after

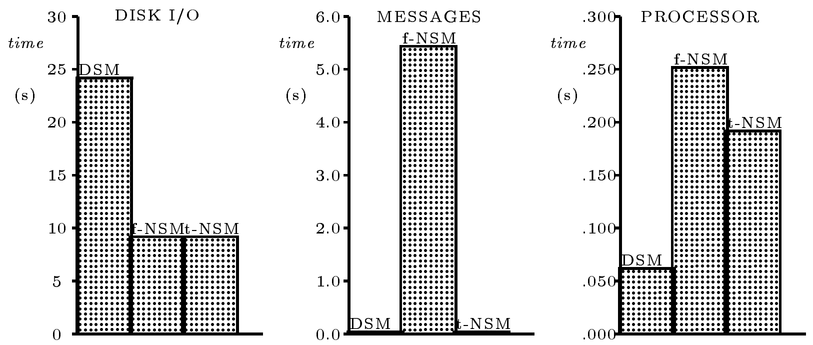

Figure 10: I/O, message and CPU times for the query of example 3 ('value-based' join).

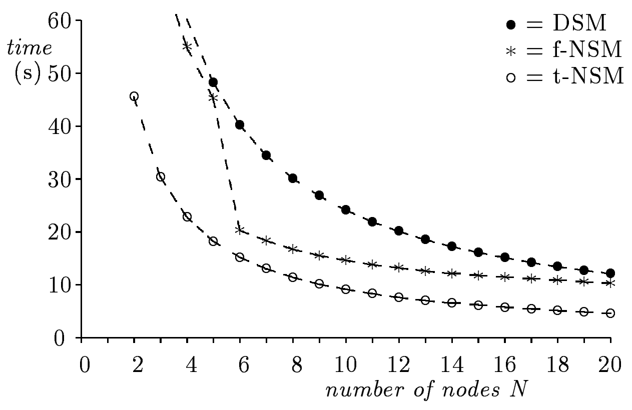

Figure 11: Total response time while varying the number of nodes with the query of example 3

redistribution. However, even for this situation DSM still has a larger response time.

Concluding, for this example query the costs are minimal with a normalized storage model with intra tuple-object clustering. A normalized storage model without intra tuple-object clustering suffers very much from communication overhead. A direct storage model suffers from the necessity to input entire tuple-objects.

6 Comments on the results and conclusions

We investigated the performance of three different storage models for complex objects, based on the possible join queries for complex objects. Attention was paid to disk I/Os, network messages and processor loads. In order to be really able to determine the best overall storage structure, other queries (and in particular updates!) need to be considered as well. Also, other storage structures or in-between storage structures may need attention. Using an index on a non-identifier attribute might give new insights as well. Nevertheless, some main conclusions can be drawn.

The direct storage model (DSM) is the best storage model with regard to network communication and processor load. The retrieval of an entire tuple-object is very fast. However, the retrieval of a single attribute of a tuple-object is not well supported. Consequently, a small and simple operation on a large tuple-object may degrade the overall query performance enormously.

A normalized storage model is much more flexible with regard to disk I/Os. In general it uses (in particular with intra tuple-object clustering) the fewest disk accesses. Only when tuple-objects need to be fetched from disk in their entirety a direct storage model is better. Although joins will be needed to reconstruct a tuple-object from its normalized representation, tuple identifiers seem to support such a reconstruction very well, provided that enough main memory space is available. However, without intra tuple-object clustering (not clustering all data of a single tuple-object on a single node) the communication overhead becomes unacceptable.

Concluding, a materialized join seems to be best supported by a direct storage model. A functional join may be well supported by DSM as well. However, since a normalized storage model with intra tuple-object clustering (t-NSM) is more flexible (e.g. in finding the identifiers themselves) and gives reasonable performance in tuple-object retrieval (and assembly), such a storage model may be preferable. For a value-based join the same conclusion can be drawn since the operations that are specific for such a join do not favour a particular storage model. So, a normalized storage model with intra tuple-object clustering seems to give the best overall performance. But, of course this may depend on the overall database load and system characteristics. In particular, since in non-standard database environments the materialized joins seem to be important, a direct storage model (DSM) might be the best as well.

The fact that we showed the results for disk I/O, network messages and processor costs as being independent of each other makes our results easily adaptable for other database systems (provided they are shared nothing). Actually one only has to pay attention to the relative importance of disk I/O versus network messages versus processor load and might change the conclusions correspondingly. Moreover, a close look at the results shows that for our parameters CPU costs seem to be unimportant whereas the disk I/O costs are a bottleneck[4]. Therefore, the conclusions based on our parameters may be interesting for many systems since in the parallel database world there is a general consensus that CPU costs are no bottleneck and that the I/O bandwidth is a severe bottleneck [4].

References

[1] J. Banerjee, W. Kim and K-C. Kim, Queries in Object-Oriented Databases, Proceedings Fourth International Conference on Data Engineering, Los Angeles, CA, Feb. 1–5, 1988 (IEEE Computer Society Press, Washington, DC, 1988) 31–38.

[2] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve and J. B. Rothnie, Jr., Query Processing in a System for Distributed Databases (SDD-1), ACM Trans. Database Syst. 6 (1981) 602–625.

[3] H. M. Blanken and A. Ybema, Storage of Versioned Objects in a CIM Environment, Proceedings International Conference on Data and Knowledge Systems for Manufacturing and Engineering, Hartford, CT, Oct. 19–20, 1987 (IEEE Computer Society Press, Washington, DC, 1987) 65–74.

[4] D. J. DeWitt and J. Gray, Parallel Database Systems: The Future of Database Processing or a Passing Fad?, ACM SIGMOD Record 19 (1990) 104–112.

[5] U. Deppisch, H. -B. Paul and H. -J. Schek, A Storage System for Complex objects, in: K. Dittrich and U. Dayal(Eds.), Proceedings 1986 International Workshop on Object-Oriented Database Systems, Pacific Grove, CA (IEEE Computer Society Press, Washington, DC, 1986) 183–195.

[6] T. Härder, An Approach to Implement Dynamically Defined Complex Objects, in: P. America(Ed.), Parallel Database Systems. Proceedings PRISMA Workshop, Noordwijk, The Netherlands, Sept. 24–26, 1990 (Springer-Verlag, Berlin, 1990) 71–98.

[7] R. Lorie and W. Plouffe, Complex Objects and their Use in Design Transactions, IEEE 1983 Proceedings of Annual Meeting – Database Week: Engineering Design Applications, San Jose, CA (IEEE Computer Society Press, Washington, DC, 1983) 115–121.

[8] P. Pistor and F. Anderson, Designing a Generalized NF^2 Data Model with an SQL-Type Language Interface, in: W. Chu, G. Gardarin, S. Ohsuga and Y. Kambayashi(Eds.), Proceedings of Twelfth International Conference on Very Large Data Bases, Kyoto, Japan, Aug. 25–28, 1986 (Morgan Kaufmann Publishers, Los Altos, CA, 1986) 278–285.

[9] P. Pistor and P. Dadam, The Advanced Information Management Prototype, in: S. Abiteboul, P. C. Fischer and H. -J. Schek(Eds.), Nested Relations and Complex Objects in Databases (Springer-Verlag, Berlin, 1989) 3–26.

[10] H. -J. Schek and M. H. Scholl, The Relational Model with Relation-Valued Attributes, Inf. Syst. 11 (1986) 137–147.

[11] E. J. Shekita and M. J. Carey, A Performance Evaluation of Pointer-Based Joins, in: H. Garcia-Molina and H. V. Jagadish(Eds.), Proceedings of ACM-SIGMOD 1990 International Conference on Management of Data, Atlantic City, NJ, May 23–25, 1990 (ACM Press, New York, NY, 1990) 300–311.

[12] W. B. Teeuw and H. M. Blanken, Representing Complex Objects in a Distributed Database: A Performance Evaluation, Universiteit Twente, Technical Report INF-90-92, Enschede, The Netherlands, Dec. 1990.

[13] W. B. Teeuw and H. M. Blanken, Joining Distributed Complex Objects: Definition and Performance, Universiteit Twente, Technical Report INF-91-35, Enschede, The Netherlands, May 1991.

[14] P. Valduriez, Join Indices, ACM Trans. Database Syst. 12 (1987) 218–246.

[15] P. Valduriez, S. Khoshafian and G. Copeland, Implementation Techniques of Complex Objects, in: W. Chu, G. Gardarin, S. Ohsuga and Y. Kambayashi(Eds.), Proceedings of Twelfth International Conference on Very Large Data Bases, Kyoto, Japan, Aug. 25–28, 1986 (Morgan Kaufmann Publishers, Los Altos, CA, 1986) 101–110.

[4]With the possible exception of a normalized storage model without intra tuple-object clustering. Then the network messages are the bottleneck. However, this storage structure has been found to be inferior.