Joining distributed complex objects: Definition and performance*

Wouter B. Teeuw and Henk M. Blanken

University of Twente, Department of Computer Science, Information Systems – Databases, P.O. Box 217, NL-7500 AE Enschede, Netherlands

Abstract


The performance of a non-standard distributed database system is strongly influenced by complex objects. The effective exploitation of parallelism in querying them and a suitable structure to store them are required in order to obtain acceptable response times in these database environments where performance tends to be critical. In this paper we use an hierarchical complex object model with object references and define several join operations for these complex objects which we will call tuple-objects. These join queries for tuple-objects can be categorized into materialized, functional and value-based joins. An analytical performance evaluation gives insight into which distributed storage structures for tuple-objects are most effective with regard to executing the different kinds of tuple-object joins.

Keywords. Complex objects; distributed databases; join operation; performance aspects; storage structures.

1. Introduction

In the so-called non-standard database application areas, such as robotics, cartography and CAD/CAM, data objects are used that are both highly structured and large in size. These large clusters of structured data are named complex objects. Complex objects are a unit of manipulation. In general, they are built by applying various complex object constructors (such as tuple, set and list) to other complex objects or basic values (such as boolean, character, integer, real, string). Also, complex objects have some notion of object identity and they have relationships among each other.

Complex objects are one of the features of object-oriented databases [2]. The structural aspects of complex objects can easily be captured in object-oriented data models [1]. But, except for rich data structuring capabilities, in non-standard database applications a high performance is generally required as well. Therefore, complex objects need to be stored in such a way that retrieving and processing them can be handled efficiently. Also, parallel processing and parallel disk I/O might be used to reduce response times.

Obviously, a pure relational storage structure without special provisions is not very well suited to manage complex objects. In particular, 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 database relations. Those time and resource

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

Correspondence to: W.B. Teeuw, University of Twente, Department of Computer Science, Information Systems – Databases, P.O. Box 217, NL-7500 AE Enschede, Netherlands.
consuming joins cannot be accepted if performance is critical. Therefore, several alternative techniques have been proposed to store complex objects.

Extensions of the relational model may keep the advantages of relational storage, while at the same time providing a better performance. Some examples are the NF$^2$ [24] and extended NF$^2$ [22] data models that use nested relations to represent complex objects. These nested relations might still be stored in (pseudo) relational storage structures, where additional join indices [29, 30] or tuple identifiers [5, 20] keep the nested structure. Alternatively, these nested relations (or in general complex objects) may be stored contiguously on disk in a single file [12, 23]. In general, both the tuple-object’s data and its link and structure information (e.g. in a header) will be stored on such a contiguous part of the disk. Finally, complex objects may be implemented in a completely non-relational way. An example is provided by the PRIMA database system [15], in which complex objects are modelled as a ‘molecules’ consisting of ‘atoms’ that are linked together in a network [14].

The choice for a particular complex object storage structure will depend on the workload on the system. The workload depends on the application, but in general complex objects tend to be retrieved in their entirety [18]. Therefore, most researchers consider object retrieval as a basis for the performance evaluation of complex object storage structures [26]. Little or no attention is paid to further processing. At most, attention is paid to the effect of set processing on complex object retrieval [16]. In this paper we will investigate how a post-retrieval processing of complex objects may influence the choice for a complex object storage structure. In particular, focus on the join operation.

In a complex object environment, three kinds of joins can be distinguished. Similar to the relational join, a complex object join might restrict the instances of one complex object type on the basis of the values of an attribute of another complex object type occurrence. However, this traditional relational value-based join seems to play a less dominant role in non-standard environments. More important are object accesses along references leading from one object instance to another. Some authors call this kind of object traversal implicit or functional joins [8]. Finally, what used to be a join in the relational system might simply be the projection on some (nested) attributes of the complex object, i.e. the join has been materialized in the complex object structure. In a non-standard environment the materialized and functional joins seem to be the most important ones. For each kind of join, we will show which complex object storage structures is almost suitable.

We focus on a distributed shared-nothing database system based on a local area network. A number of homogeneous nodes communicate with each other by message passing over the network. The network is the only resource they share. Each node consists of a processor, some internal memory and a disk drive. Each node owns a disjunct portion of the database data. Distributed systems are more and more used because they increase reliability, availability, and in particular performance. Moreover, shared-nothing systems have demonstrated speedup and scaleup to hundreds of nodes and seem to be the basis for distributed database technology [11]. We examine three storage structures, two of which are normalized. Given a join query, a storage structure, and some parameter values, we will show how to estimate the costs of query execution. In this way, our performance evaluation may, together with identical results for selection queries, be the base for a query optimizer and/or a data allocation manager for complex objects in a distributed database system. Our join queries have complex objects as input and (structured) values as output. So the results are considered as data rather than objects. 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.

The remainder of this paper is organized as follows. In section 2 we describe what a complex object is. Therefore, we introduce the notions of tuple-objects and composed...
Joining distributed complex objects

objects. A tuple-object is a kind of extended 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, but rather provide a view on complex objects which enables performance evaluation. How to represent tuple-objects in different storage structures is discussed as well. Some examples of tuple-objects, which will be used during the rest of the paper, are presented. In section 3 we explain what it means to do a join on tuple-objects. Also, some example queries are presented. In section 4 we show how to build the result of a join query from the (nested) tuples that are stored in the distributed database. For each kind of join, a single example query is exhaustively analysed. The query (expressed in an extended version of SQL) is translated into an optimized query tree (algebra). Also, we show how, from this algebra, response times can be calculated. In section 5 we present the results of our analytical performance evaluation. In section 6, finally, some conclusions are presented.

2. Complex objects

2.1. Tuple-objects and composed objects

In this section we will describe our vision on a complex object, which we call a tuple-object. A tuple-object is constructed from some basic types by applying three constructors: the tuple, set and list constructor. Among the basic types (such as boolean, character, integer, real, string) are an identifier type and a reference type as well. The constructors are completely orthogonal (i.e. they can be applied in any order), except that 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 internal structure. Their attributes may differ in value but not in structure. Connected to each tuple-object is a single unique system generated identifier that distinguishes the object from all others [17]. The identifiers can be used by tuple-objects to refer to each other.

Figure 1 shows an example tuple-object of the type Doctor. The tuple-object is a tuple containing four attributes: two atomic attributes (identifier and name) and two non-atomic attributes (the tuple attribute private and the set attribute disease). 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 disease is a set of atomic values. In the figure, the symbols ( ) denote a tuple, { } a set and [ ] a list. Each tuple-object has an attribute identifier of the basic type OID (Object IDentifier). The value of this attribute is unique and it can't be affected by user updates. The attribute of the type REF(Disease), a reference type, can be seen as an attribute of type OID whose attribute values have to obey a referential integrity

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

Fig. 1. Declaration of a Doctor tuple-object.
constraint: they have to be the identifiers (i.e. value of the identifier attribute) of tuple-objects of the type called Disease.

A tuple-object is hierarchically structured and resembles the NF\(^2\) [24], and in particular the extended NF\(^2\) [22] data model. The characteristic property of a tuple-object is its existence dependency. That is, 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 references between them. Starting from a specific root tuple-object, the whole composed object can be retrieved by following these links. Again, we may consider types and instances. A composed object type is specified by the corresponding tuple-object types and their relationships. A composed object instance is a collection of tuple-object instances referring to each other. Composed objects may have an hierarchical, network-like or recursive structure. A composed object may be part of another (larger) composed object as well. Several composed objects may share a single tuple-object, even though not stored redundantly.

A composed object may be compared with the concept of a view in a relational database. A composed object is derived from the tuple-objects in the database just as a view is derived from the base relations in the relational model. The tuple-objects in a composed object are not existence dependent. The deletion of some of the tuple-objects in a composed object does not necessarily imply the deletion of all the remaining tuple-objects this composed object consists of. Composed objects resemble the molecules in the MAD data model [14].

In the next subsections we show how tuple-objects can be implemented in the database. In section 2.2 we examine a direct mapping of tuple-objects onto the storage structures of the database. In section 2.3 we focus on normalized storage structures for tuple-objects. Finally, in section 2.4, we pay some attention to the operations on tuple-objects. In section 3 we will continue on this subject by examining the join operations for tuple-objects.

2.2. A direct implementation of tuple-objects

2.2.1 The direct storage model (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 (i.e. complex objects) as a whole was originally called direct storage model [30] and is sometimes called flattened storage model [13]. We will refer to this model as direct storage model or 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 part of a tuple-object or of single attribute value may be inefficient since possibly the whole tuple-object has to be retrieved. Figure 2 shows the direct representation of the Doctor tuple-object that has been defined in Fig. 1. Figures 3 through 5 show yet three other...
example tuple-objects which we will use throughout this paper. Notice that all occurrences of the tuple-objects of the same type form a single nested relation, which we will refer to as the DOCTOR, PATIENT, ILLNESS, and HOSPITAL relation respectively.

Several implementations for DSM exist. In the Darmstadt database kernel system (DASDBS) project [12, 21], a tuple-object (called a complex record) is implemented as a header and a body. The header contains the page and structure information. The body contains the tuple-object’s data. If the tuple-objects are smaller in size than a page, several tuple-objects (of the same type) will be stored on a single page, but a tuple-object will not span pages. A large tuple-object will be stored on a sequence of pages. These pages will not be shared with other tuple-objects. The first page contains all the necessary page information and probably the whole header. Thus fetching the whole page set of a tuple-object can be avoided if only parts of it are required. In the Advanced Information Management Prototype (AIM-P) project [23] a tuple-object (called NF² tuple) is stored as a set of ‘tuples’ containing either data or link information. All these ‘tuples’ of a single tuple-object, which are linked together by a pointer mechanism, will be stored clustered together on disk. In EXODUS [6, 7], finally, all schema information is kept out of the storage system and handled by higher levels of the system instead. The storage management component of EXODUS handles a tuple-object (called storage object) as being an uninterpreted byte sequence of virtually unlimited size. Small tuple-objects reside on a single disk page. For large tuple-objects, the tuple-object identifier points to a ‘large-object-header’, which resides on a page with other large-object-headers and small tuple-objects. The large-object-header contains pointers to pages which, again, contain pointers to pages etc., finally pointing to leaf pages with data.

In our performance evaluation it does not matter which implementation for DSM we choose, provided some assumptions are obeyed. We assume that the tuple-objects are randomly distributed over the nodes, with each node storing an equal number of tuple-objects of each type. Per node, the tuple-objects of a single type are stored, one behind another, in a single sequence of pages (file). In order to account for the disk space needed
for structure information, as well as some wasted disk space (e.g. for tuple-objects that start at the beginning of a next page rather than at the end of a previous page), we assume the page occupation to be less than 100%. Finally, we assume that an index on the tuple-object identifier exists. It enables us to determine the necessary page information given the tuple-object identifier.

2.3. A normalized implementation of tuple-objects

2.3.1 The normalized storage model (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, the so-called tuple-identifiers or tids, are used to store the tuple-object structures, as well as to support fast reconstruction. For instance, a tid can be used as a pointer. Then the parent tuple may contain the tids of its child tuples [3]. Also, additional (binary or nested) relations whose attributes values are tids may be maintained in addition to the relations with tuple-object data. These so-called binary [29] or hierarchical [30] join indices are structured in such a way that they keep the tuple-object structure. Alternatively, the tid itself may be constructed in such a way that it contains information about its root and parent tuples [5, 20]. The latter approach is the one we use. The advantage of this approach is that, since each tuple has only a single root and parent tuple, a constant storage space per tuple is needed.

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 tids 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 of tuple-object retrieval. An example will make NSM more clear.

Figure 6 shows the NSM 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 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 tids in the DOC_DISE relation have no parent part since on the first level of nesting the parent is equal
Joining distributed complex objects

Fig. 6. Normalized representation of the Doctor tuple-object.

to the root. The normalized representations of the Patient, Illness, and Hospital tuple-objects are shown in Figs. 7–9.

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). Figure 10 shows how an example tuple-object of the type Doctor may be distributed over the nodes with f-NSM and t-NSM. The figure illustrates the use of the tid attributes as well. In both cases, each node stores a fragment of each flat relation. Per node, such a relation fragment will be stored in a single separate file (sequence of pages). The pages of this file contain some wasted space. An index on the root part of the tid exists. This index is not necessarily dense. Per relation fragment, one entry per tuple-object instance or per page may be sufficient.

2.3.2 The binary storage model (BSM)

In a binary storage model (BSM)\(^1\) all atomic attributes are stored in a separate binary relation, whose attributes are a tuple identifier and a value respectively. Figure 11 shows an example for the Doctor tuple-object. As with NSM, a binary storage model is normalized. Therefore, a set is considered as a relation and a list as an ordered relation. BSM is very well suited for the storage of very large attributes (long fields). The selection and projection on a few attributes will be efficient. In general BSM reduces update performance. In this paper, we will not consider BSM any further since, as compared with NSM, we do not expect to gain new insights by evaluating BSM.

\(^1\)Sometimes this storage model is called decomposed storage model [30].
2.3.3 The traditional relational model for tuple-objects

Tuple-objects can also be mapped on a traditional relational model (without special provisions like the tids of NSM). Figure 12 shows a possible representation of the Doctor tuple-object in the traditional relational model. Tuple-objects of the type Doctor are stored in two flat relations, referred to as D1 and D2. Since the relational model doesn’t know identifiers, the attribute identifier has been left out. We assume that a doctor is uniquely determined by his name and address. Therefore, the attributes name, street, nr and city together form a key to a tuple-object of the type Doctor. The attribute disease is a string. It contains the name of a tuple-object of the type Illness. We assume that the name of an illness can be used as a unique key. The reason why we did not store all atomic attributes in a single large relation was to prevent the replicated (redundant) storage of attributes as much as possible. For, the number of tuples in such a single relation would be the product of the extensions of D1 and D2.
Joining distributed complex objects

Figure 12 clearly shows the shortcomings of the traditional relational model. The concept tuple-object is unknown to the relational model. Therefore the semantic data structure of a tuple-object is kept by a combination of user information (expressed in the queries), data dictionary information and primary/foreign key implementations. In general, a tuple-object will be represented by several tuples of several relations. Therefore, join operations will be required to reassemble the tuple-object from the corresponding database relations. In the traditional relational model these join operations will be time and resource consuming because the helping t.id attributes are missing. Moreover, a key may consist of several attributes which makes the re-assembly joins even more expensive, enlarges the indices, and causes much redundant storage. Constructing and sorting the traditional relations as with NSM makes the keys even longer, and does not solve all the performance problems. As a consequence, the retrieval and assembly of entire tuple-objects will be rather slow. Since we do not expect a better performance of the relational model (as compared with for instance NSM), we do not consider it any further.

2.4. Processing tuple-objects

2.4.1 Set of operators

Since a tuple-object is a 'tuple', a set of tuple-objects of the same type form an extended non-first-normal-form 'relation'. For such relations we may, analogous to the traditional relational model, define an algebra. We will show some operations. All input operands of these operators are tuple-objects. We consider the output of these operators as pure data (inclusive the identifier attributes). So if the result has to be stored in the database, a new tuple-object identifier has to be added first. Of course a different philosophy (like considering everything as an object) is possible. However, such issues are irrelevant with regard to the performance analysis presented in this paper.

Select The selection \( \sigma_x \) selects whole tuple-objects based on a predicate \( x \) to be applied to a tuple-object.

Project The projection \( \pi_x \) extracts specific attributes (indicated by the set \( x \)) of all tuple-objects in a 'relation' and eliminates duplicates. Obviously, if the attribute containing the tuple-object identifier is in \( x \), there will be no duplicates.

Dot The dot \( T \cdot a \) extracts a single specific attribute \( a \) of a tuple-object \( T \). As opposed to the projection, the top-level tuple constructor is removed. Moreover the result is, if necessary, transformed to a (set of) tuple(s) by considering set elements (as well as list elements) as tuples. Thus, if \( a \) is an atomic attribute the dot operation has the same effect as the projection. If \( a \) is a set attribute (and identically for list attributes), the result of the dot operation is this set if the set elements are tuples, and the result is this set with each set element being transformed to a unary tuple if the set elements weren’t tuples.

Product The Cartesian product \( R \times S \) builds a 'relation' from two relations \( R \) and \( S \) consisting of all possible concatenated pairs of tuple-objects, one from each of the two
relations. Notice the result is considered as data. So we ignore problems like each result tuple has two tuple-object identifiers. As the result is a relation (in the traditional relational sense), we assume the Cartesian product of two sets of tuples (relations) \( \{(a_1, a_2)\} \times \{(b_1, b_2)\} \) has as result \( \{(a_1, a_2, b_1, b_2)\} \) instead of \( \{((a_1, a_2), (b_1, b_2))\} \).

**Union** The union \( R \cup S \) builds a ‘relation’ consisting of all tuple-objects appearing in either \( R \) or \( S \).

**Intersection** The intersection \( R \cap S \) builds a ‘relation’ consisting of all tuple-objects appearing in both \( R \) and \( S \).

**Difference** The difference \( R \setminus S \) builds a ‘relation’ consisting of tuple-objects appearing in \( R \) and not in \( S \).

**Join** The join \( R \bowtie_x S \) applies a selection based on a predicate \( x \) on the Cartesian product of two relations \( R \) and \( S \).

**Semijoin** The semijoin \( R \bowtie_x S \) is identical to the join with only the attributes of relation \( S \) participating in the result.

**Divide** The division \( R \div S \) of a binary relation \( R \) by a unary relation \( S \) builds a unary relation consisting of all values of one attribute of \( R' \) that match (in the other attribute) all values in \( S \). \( R' \) is the binary relation that arises if \( R \) is un-nested. Notice that, since \( R' \) must be binary, the attributes of \( R \) have to be atomic or unary tuples, or sets/lists of atomic values or unary tuples.

Since a relational tuple is a special case of a tuple-object, these operations hold for sets of tuples (traditional relations) as well. Neglecting the operations that have no sense in a traditional relational environment (i.e. the DOT operation), the tuple-object operations appear to be the normal operations of the relational algebra as, e.g., defined by Date in chapter 13 [9]². Consequently, we can use these operations for tuple-objects to build results from the basic (nested) relations, not only for the direct storage model, but also for the normalized storage model. We will do so in section 4.

**Figure 13** shows some examples of algebraic tuple-object operations. The natural join is an equi-join which includes the join attribute only once in the result relation. The semijoin rather selects tuples of one operand relation based on the join predicate. So the attributes of the other operand relation are removed from the result. Notice that (in this figure) the first operand of the division operator is already in an un-nested form.

### 2.4.2 Ordering relation

To be able to execute joins or selects we must be able to compare attributes with each other or with a constant. However, in a tuple-object the attributes may be not only atomic, but also tuples, sets or lists. To test (complex) attributes to be equal we need a linear order ‘\(<\)’ that obeys the laws of trichotomy (exactly one of \( a < b, a = b \) and \( b < a \) is true) and transitivity (if \( a < b \) and \( b < c \) then \( a < c \)).

Many alternative orderings for nested tuples, sets and lists are possible. However, we are

![Fig. 13. Some tuple-object algebra operations.](image)

²Actually, we have suitably generalized Date’s relational algebra operators for tuple-objects.
only interested in the fact that an ordering exists and therefore we will just show one such ordering which seems to be appropriate for a tuple-object environment [19]. It is a recursive transitive ordering based on cardinality of sets and lexicographical ordering. That is, the elements of the powerset of \( \{1, 2, 3\} \) are ordered \( \emptyset < \{1\} < \{2\} < \{3\} < \{1, 2\} < \{1, 3\} < \{1, 2, 3\} \). Tuples and lists are compared starting from the first attribute or element respectively.

Suppose \( \text{CARD}(S) \) denotes the cardinality of a set \( S \), \( \text{MIN}(S) \) the smallest element of a set \( S \) and \( \text{HEAD}(L) \) the first element of a list \( L \). Then, for objects \( A \) and \( B \) of the same type (that is both atomic, both a set or both a list) \( A < B \) if:

- \( A \) and \( B \) are atomic and \( A < B \),
- \( A \) and \( B \) are sets and either \( \text{CARD}(A) < \text{CARD}(B) \)
  or \( \text{CARD}(A) = \text{CARD}(B) \) and \( \text{MIN}(A) < \text{MIN}(B) \)
  or \( \text{CARD}(A) = \text{CARD}(B) \) and \( \text{MIN}(A) = \text{MIN}(B) \) and \( A \setminus \text{MIN}(A) < B \setminus \text{MIN}(B) \),
- \( A \) and \( B \) are tuples (respectively lists) and either \( \text{HEAD}(A) < \text{HEAD}(B) \)
  or \( \text{HEAD}(A) = \text{HEAD}(B) \) and \( A \setminus \text{HEAD}(A) < B \setminus \text{HEAD}(B) \).

This linear order relation ‘\(<\)’ makes it possible to formulate those selection and join predicates which are thought necessary for a tuple-object environment.

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. What actually happens is that tuples of the one relation are selected based on attribute values in the other relation (primary/foreign key). 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. Some authors call this kind of object traversal a join as well [8] since these implicit or functional joins also relate basic units to each other.

In general, a join for tuple-objects is an operation that correlates different (complex) attributes of arbitrary tuple-object types [18]. In this section we examine joins for tuple-objects. We will make a distinction between three different types of joins. In all cases, we consider the join results as values (rather than tuple-objects).

Materialized join Since a tuple-object has an hierarchical structure, operations that would be a join in the traditional environment might simply be the retrieval of some (or possibly a single) attributes in the tuple-object model. The join has been materialized in the tuple-object structure.

Functional join Tuple-objects have been connected to each other by links. Following these links between tuple-object instances conceptually means joining them (so-called functional or implicit joins).

Value-based join 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. As opposed to the traditional relational model, this predicate may contain tuple, set and list conditions.

In the next subsections we will show some example queries for each of these kinds of joins. In the example queries, the tuple-object types as described in section 2 will be used. In section 4 the example queries will be used for our analytical performance evaluation. The example queries are expressed in an extended-SQL query language. Explanation of the query language will only be given if the query might not be clear.
3.1. Materialized joins

A materialized join is a query that would be a join in the relational model, but simply becomes the retrieval of (an) attribute(s) with tuple-objects. The join has been materialized in the tuple-object structure. We will illustrate the materialized join by an example.

**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

Functional joins are the navigation from tuple-objects to tuple-objects by following the links between them. In the next example a functional join is shown.

**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.

This 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'. This part is comparable with the (traditional) relational selection. Second, for each Hospital tuple-object that is found we have the selection of a number of tuple-objects of the type Doctor where the identifiers of these tuple-objects are in the doctor attribute of the corresponding 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 known since they are given by the value of an attribute of another tuple-object instance. Together, we have a functional join.

With a functional join, the join attributes are always tuple-object identifiers or tuple-object references (thus of the type OID or REF(X)). Consequently, 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

A value-based join is, analogous to the relational join, the Cartesian product of two sets of tuple-objects with a selection on the result. The selection predicate tests the attribute values of the tuple-objects and may contain tuple, set and list conditions. We will present some examples.

\(^{3}\)With NSM, probably the so-called t.id attributes might be involved in functional joins as well.
3.3.1. Atomic valued join attributes

The next example shows a join query in which the join predicate consisting of four conditions, each one testing attributes to be equal (equi-join). Other conditions could have been included as well, e.g. a greater than (>) test.

Example 3. We want to check which persons are a doctor as well as a patient, assuming that a person is determined by its name and address.

```
SELECT {(d.name, d.private.address)}
FROM d IN Doctor,
    p IN Patient
WHERE d.name=p.name AND
    d.private.address.street=p.private.address.street AND
    d.private.address.nr=p.private.address.nr AND
    d.private.address.city=p.private.address.city
```

The output of this query is a set of binary tuples. The first attribute contains the name of a person who is a doctor as well as a patient. The second attribute is a tuple attribute containing its address information. Obviously, in the SELECT clause d.X may be replaced by p.X without affecting the result.

The first simple example strongly resembles a traditional relational join. There is, however, an important difference. In the example, the atomic join attributes in a single tuple-object are not part of the same ‘subrelation’ of the hierarchically structured tuple-object. The attribute name is part of the root tuple (of both Doctors and Patients), the other attributes are on a nested level. In section 4 we will show that the fact whether the join attributes in a (single) tuple-object are part of the same sub-structure or not may have an influence on the performance of the join query. Such problems do not occur in the first normal form relational model.

3.3.2. Tuple valued join attributes

The query of Example 3 can be expressed in a more convenient way since the attributes street, nr and city make all the attributes of the tuple constructed attribute address.

Example 4. As example 3, but using a shorter notation.

```
SELECT {(d.name, d.private.address)}
FROM d in Doctor,
    p in Patient
WHERE d.name=p.name AND
    d.private.address=p.private.address
```

In this join, one of the join attributes is a tuple constructed attribute. When executing this join, tuples of atomic values will be tested to be equal rather than atomic values themselves. The attributes in such a tuple will be scanned one by one, meanwhile comparing them with their counterpart in another tuple.

In the example query we have an equi-join on tuple attributes. Moreover, these tuples contain atomic attributes. However, since an ordering for tuples exists (section 2.4.2), more complex join conditions on tuples may be defined as well, inclusive non-equijoins. In particular, the tuple structured join attribute may contain non-atomic attributes as well. Then the process of testing tuples to be equal becomes a recursive one. In such a case, the join on the tuple attribute may conceptually be translated to a join on atomic and set
attributes that are not necessarily on the same level of nesting. A join on a set attributes is shown in the next example.

### 3.3.3 Set valued join attributes

**Example 5.** We want to know whether two diseases have the same symptoms. The output is a set of binary tuples, each tuple containing a pair of illness names.

```sql
SELECT { (i1.name, i2.name) }
FROM i1 in Illness,
    i2 in Illness
WHERE i1.symptom = i2.symptom AND
    i1 ≠ i2
```

In the example the join attribute is the set attribute `symptom`. Therefore sets have to be tested to be equal, for example by first testing the cardinalities to be equal (notice that a set does not contain duplicates) and then testing each element of the one set to be in the other set as well. Since elements of a set may be non-atomic again, in general the comparison of sets is a recursive process. Notice that we can only compare those sets with each other which contains elements of the same type. We can’t compare a set of integers with a set of characters or (even not) with a set of unary tuples containing an integer.

The join on a set attribute differs from the joins mentioned before since with sets the conditions in the join predicate may contain operators like SUBSET (\(\subseteq\)), SUPERSET (\(\supseteq\)), IN (\(\in\)), etc. Fortunately, from a conceptual viewpoint there is no essential difference between using these set inclusion operators and using a linear order relation ('\(<\)'). The next example shows the use of set inclusion in the join predicate.

**Example 6.** 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.

```sql
SELECT {d.name}
FROM d IN Doctor,
    p IN Patient
WHERE d.disease \(\supseteq\) (SELECT {i.disease}
    FROM i IN p.illness
    WHERE p.name = 'Brown')
```

As we will explain later on, the latter query is what we call the DIVIDE operation in the traditional relational algebra.

Finally a last example, which closely resembles Example 6.

**Example 7.** We want to find the names of the doctors who have been specialized in at least one of the diseases Mr. Brown suffers from. The output is a set of doctor names.

```sql
SELECT {d.name}
FROM d IN Doctor,
    p IN Patient
WHERE EXISTS X IN d.disease: X IN (SELECT {i.disease}
    FROM i IN p.illness
    WHERE p.name = 'Brown')
```

*Actually we already used the IN operator in Example 2, though that was just a simple non-recursive case.*
Join predicates which contain set inclusion operations are unknown to the traditional relational model. However, testing (one by one) the elements of a set to be included in another set can easily be done using a linear ordering relation '<'. Therefore, though performance aspects may be different, from a logical point of view comparing sets is not different from comparing atomic values.

3.3.4 List valued join attributes

Joining on a list attribute involves comparing lists with each other. With an equi-join, the elements of corresponding lists have to be scanned and compared with each other one by one. Non-equi-joins use an ordering for lists to evaluate the join predicate (section 2.4.2). With a list of non-atomic attributes, recursion (nesting) shows up again. There is one particular difference with the joins mentioned so far: the join predicate may contain list specific operations like HEAD and TAIL. As with the set specific operations, in essence these operations do not differ from a linear order relation '<'.

Example 8. We want to find doctors who work in the same team. We assume that the list attribute phone of the Doctor tuple-objects has as first element a private phone number whereas the other elements are hospital/department numbers in a fixed order. Doctors are supposed to be in the same team if the latter numbers are equal.

\[
\begin{align*}
\text{SELECT } & \{ (d1.\text{name}, d2.\text{name}) \} \\
\text{FROM } & \text{d1 IN Doctor,} \\
& \text{d2 IN Doctor} \\
\text{WHERE } & \text{TAIL(d1.\text{private.\text{phone}}) = TAIL(d2.\text{private.\text{phone}}) AND} \\
& \text{d1} \neq \text{d2}
\end{align*}
\]

The output of this query is a set of binary tuples representing pairs of doctors who are in the same team.

Notice that only a single tuple-object type is involved in the join query of Example 8.

3.3.5 Alternative valued join attributes

With basic valued join attributes, the conditions in the join predicate involve simple tests like =, <, >, etc. With sets and lists appearing in the join predicate, set and list operations turn up as well. In general, depending on the other (possibly user defined) types existing in the data model, many other comparison tests may appear in the join predicate.

To take just one example, suppose the tuple-objects of the type Doctor or Patient contain an attribute photograph of the type IMAGE. This attribute might, e.g. be an 8 Kbyte string of 0's and 1's, representing a black(0)-white(1) picture of 256 by 256 pixels. Such kind of attributes, which are called long fields, are likely to appear in non-standard database systems. With an attribute of the type IMAGE as join attribute, a join condition may be very complex. For example, the join condition might test whether two photograph attributes represent the same person. Evaluating such a join condition will require all kinds of very complex image processing and pattern recognition operations.

However, although the tests in a join condition may be arbitrarily complex, the basic join execution strategies remain the same. It is just a value-based join.

4. Performance evaluation

In Section 3 we introduced several kinds of joins for tuple-objects. In this section, we show how we can analytically evaluate the performance of these joins for the DSM, f-NSM and
t-NSM storage models. The results of such a performance evaluation will be presented in section 5.

This section consists of two parts. First, we show how to translate the extended-SQL example queries to algebraic formulas for a distributed environment. These algebraic formulas, based on the operators of section 2.4.1, show how the results of the example queries can be constructed from the base 'relations' in the database. The assumptions on which the formulas have been based will be presented as well. Second, we show how the response times for these example queries can be calculated from these formulas. Also, default system and tuple-object parameter values will be presented.

Given a certain join query, this section shows how to estimate the performance of the query for each storage structure (DSM, f-NSM, or t-NSM). In this way, it is possible to determine which storage structure is the best for a given join query. To find a best overall storage structure, a workload is needed. This workload, however, depends on the particular application that is in mind. Therefore, we have chosen one 'typical' join query for each kind of join. We will exhaustively analyse the queries of Example 1 (get name and symptoms of all illnesses, a materialized join), Example 2 (get all doctors working in the hospital 'central', a functional join), and Example 6 (get all doctors who can treat all diseases Mr. Brown suffers from, a valued-based join with a set condition in the join predicate). In this way we want to get insights into how a particular storage structure behaves for a particular class of join queries. Based on this information, we can estimate which storage structures are best for which kind of workloads.

4.1. Building query results for the different storage models

In this subsection we show how the join results are built from the basic database (nested) relations with DSM, f-NSM and t-NSM. We will present the equations that show us how successive operators have to be applied to these stored relations in order to get the desired result. The (optimized) formulas also express the possibilities for parallelism. In making these equations, we used some basic assumptions about the query execution strategies. The assumptions, listed in order of decreasing priority, are:

(a) Results are generated, not to be stored in the database, but to be shown to a user. Therefore the results are constructed in a suitable form (see point d) and sent to an output (which isn't one of the N nodes).
(b) We assume that, as much as possible, the data are processed where they have been allocated. That is, the parallelism (which should be used whenever possible) is determined by the data allocation. For example, if a node has to join two relations, we do not pay attention to all kinds of clever parallel join algorithms that redistribute the join operands over all nodes in order to gain performance [28]. Nevertheless, parallelism is used since in general this local join will, properly speaking, be the join between the local fragments of two distributed relations that are joined by a parallel join algorithm.
(c) Message passing over the network should be minimized. Reducing the network load is more important than reducing the CPU load.
(d) Since we consider the results as user data (and a user knows the concept tuple-object), with NSM the final result may be a traditional relation (Example 1), as well as a nested relation (Examples 2 and 6). Obviously, we may not expect from a user to be able to handle large tuple-objects that have been completely represented as a first normal form relation. Therefore not all results can be in relational form. Constructing the result with NSM in a pure NSM form is unsatisfactory as well, since in that case the joins which are required to reassemble a tuple-object from its normalized representation would (in many cases) still not be executed. So some results have to be represented in a direct form (as is
always the case with DSM). Notice that if such a directly represented result has to be stored in the database, a complete normalization step will be required with NSM.

(e) All relations (which are nested with DSM and flat with NSM) are equally distributed over the nodes. Of course, this is true only so far the number of tuples and the clustering (NSM) allow such a distribution. With DSM and t-NSM we assume 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.

Before we continue, some points about the notation. First, in the formulas we use the notation $X_i$ to denote the fragment of $X$ that has been allocated on node $i$, with $1 \leq i \leq N$, assuming $N$ nodes. $X = \bigcup_{i=1}^{N} X_i$ may be an arbitrary (nested) relation as shown in Figs. 2–9. Second, the symbol $\bowtie$ has to be read as if the symbol $\bowtie$ was printed. The superscript star only indicates that the join involves tuple-object identifier or tid attributes rather than 'normal' attributes. We just wanted to mark these joins since they tend to be faster than 'normal' joins. In particular, since indices on tuple-object identifiers exist (see the description of DSM and NSM), the number of page accesses will be low. Third, with tid^{(r)} we indicate the root part of a tid attribute. Finally, the final query is indicated with RESULT and an intermediate result is indicated with TEMPX.

The equations we show might have been decomposed into a number of sub-formulas. In principle, the sub-formulas can be executed in an arbitrary order, unless the result of one subformula is used in another sub-formula (which obviously forces an execution order). The addition ‘$1 \leq i \leq N$’ indicates that all nodes can execute in parallel. With a union $\bigcup_{i=1}^{N} X_i$ appearing in a formula, all nodes will execute an expression $X$ in parallel as well (whereupon the results are gathered together). A more detailed elaboration of all equations can be found in an additional report [27].

4.1.1 Direct storage model (DSM)

The result of Example 1 (names and symptoms of all illnesses) will be built like:

$$\text{RESULT1} := \bigcup_{i=1}^{N} (\pi_{\text{name,symptom}} \text{ILLNESS}_i)$$

(1)

In order to reduce communication cost, the projection is applied before the union. Obviously, all nodes process in parallel.

The query of Example 2 (doctors working in the hospital 'central') becomes:

$$\text{TEMP1} := (\sigma_{\text{name} = \text{central}} \text{HOSPITAL}_i) \cdot \text{doctor} \ (1 \leq i \leq N),$$

$$\text{RESULT2} := \bigcup_{i=1}^{N} (\text{TEMP1}_j \bowtie_{\text{doctor-identifier}} \text{DOCTOR}_i) \ (\text{TEMP1}_j \neq \emptyset),$$

(2)

$\text{TEMP1}$ is a set of doctor identifiers. After it has been generated, it is used to select the corresponding doctors (the semijoin). Since we assume that there is only a single hospital called 'central', all $\text{TEMP1}_i$ are empty except for $\text{TEMP1}_j$. Because all nodes use $\text{TEMP1}_j$, it 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, $\text{TEMP1}_j$ may (and will) be distributed (rather than replicated) over the nodes. That is, each tuple of $\text{TEMP1}_j$ is only sent to a single node (rather than to all nodes).

Finally, the query of Example 6 (doctors who can treat all diseases of Mr. Brown) becomes:
\[
\text{TEMP}_1 := \pi_{\text{disease}} \left( \sigma_{\text{name}=\text{Brown}} \cdot \text{PATIENT} \cdot \text{illness} \right) \quad (1 \leq \iota \leq N),
\]
\[
\text{TEMP}_2 := \pi_{\text{name,disease}} \cdot \text{DOCTOR} \quad (1 \leq \iota \leq N),
\]
\[
\text{RESULT}_1 := \bigcup_{\iota=1}^{N} \left( \text{TEMP} \cdot \text{TEMP}_2 \right) \quad (\text{TEMP}_2 \neq \emptyset)
\] (3)

\text{TEMP}_1 \text{ is a set of unary tuples which contain references to Mr. Brown’s diseases. In this example, again, we assume that only a single tuple-object matches the selection predicate. So there is only one Mr. Brown and consequently all \text{TEMP}_1, are empty, except for \text{TEMP}_1. \text{TEMP}_1 \text{ has to be replicated (in its entirety) on all nodes.}

4.1.2 Normalized storage model without intra tuple-object clustering (f-NSM)

Using f-NSM, the result of Example 1 (names and symptoms of all illnesses) will be built like:

\[
\text{RESULT}_1 := \bigcup_{\iota=1}^{N} \left( \pi_{\text{name,description}} \left( \text{ILL_ROOT} \cdot \text{ILL_SYM} \right) \right)
\] (4)

\text{ILL_SYM} \text{ is replicated on all nodes and in parallel the nodes construct result tuples by executing a join and a projection. Then the result is sent to the output (the outer union). Actually, \text{ILL_SYM} \text{ needs not to be replicated entirely on all nodes. Using the tuple-object identifiers stored in the \text{tid} attribute, \text{ILL_SYM} \text{ will rather be distributed over the nodes since for each tuple-object there is only one tuple in \text{ILL_ROOT}. RESULT}_1 \text{ is a flat relation.}

Alternative execution strategies might be used. For instance, one can reverse the roles of \text{ILL_ROOT} \text{ and } \text{ILL_SYM} \text{ and replicate } \text{ILL_ROOT} \text{ on all nodes first. This might be advantageous since } \text{ILL_ROOT} \text{ has to be entirely replicated on all nodes whereas } \text{ILL_SYM} \text{ can be distributed over the nodes). A disadvantage would be that the result will not be clustered, neither on illness names, nor on symptom names. The strategy of equation 4 clusters the result on illness names, which is highly desired from a user’s point of view. Alternatively, it would be possible to collect the entire relations \text{ILL_ROOT} \text{ and } \text{ILL_SYM} \text{ on a single node first, and then perform the join. Although in some cases this approach might be faster, we preferred to use as much parallelism as possible. Besides, parallel join algorithms based on hashing the operand relations over nodes have proven to be very effective [28].

The query of Example 2 (doctors working in the hospital ‘central’) becomes:

\[
\text{TEMP}_1 := \pi_{\text{identifier}} \left( \sigma_{\text{name}=\text{central}} \cdot \text{HOS_ROOT} \right) \quad (1 \leq \iota \leq N),
\]
\[
\text{TEMP}_2 := \bigcup_{\iota=1}^{N} \left( \pi_{\text{doctor}} \left( \text{TEMP}_1 \cdot \text{HOS_DOCT} \right) \right) \quad (\text{TEMP}_1 \neq \emptyset),
\]
\[
\text{TEMP}_3 := \text{TEMP}_2 \cdot \text{DOC_ROOT} \quad (1 \leq \iota \leq N),
\]
\[
\text{TEMP}_4 := \text{TEMP}_2 \cdot \text{DOC_ADDR} \quad (1 \leq \iota \leq N),
\]
\[
\text{TEMP}_5 := \text{TEMP}_2 \cdot \text{DOC_PHON} \quad (1 \leq \iota \leq N),
\]
\[
\text{TEMP}_6 := \text{TEMP}_2 \cdot \text{DOC_DISE} \quad (1 \leq \iota \leq N),
\]
Joining distributed complex objects

\[
RESULT\ 2 := \bigcup_{j=1}^{N} (TEXPII_j)_{\text{identifier}=\text{tid}(r)} \supseteq \bigcup_{i=1}^{N} TEXPI \supseteq \bigcup_{i=1}^{N} TEXPIV \supseteq \bigcup_{i=1}^{N} TEMPIV \supseteq \bigcup_{i=1}^{N} TEMPI \tag{5}
\]

The construction of TEMPI contains a projection on the identifier attribute in order to make the intermediate result as small as possible (reduction of communication cost). Due to this projection the construction of TEMPII contains a semijoin rather than a join. TEMPII is the set of identifiers indicating the requested doctors (assuming there is only a single hospital called "central"). TEMPII will be sent (in its entirety) to all nodes whereupon these nodes select (by semijoins) all needed relation tuples, with as result TEMPIII-VI. TEMPIV-VI will be redistributed over the nodes whereupon the result tuple-objects are generated in parallel on the nodes where the root-tuple (in TEMPIII) had been stored. Finally the result is collected (sent to the output).

Although not suggested directly by equation 5, we assume that the final result constructing joins (last formula) only deliver relevant information. That is, the result will be in some kind of direct representation. This is necessary since we may not expect that a user would be able to retrieve any information from a doctor tuple-object represented in a first normal form relation. The result has to be in a user-friendly form. Obviously, no t.ids appear in the result at all. Notice that first joining all doctor information into a single nested tuple per doctor, and then selecting the relevant doctor tuples would, from a performance point of view, not be a smart strategy. Then, all doctor tuple-objects have to be fetched from disk in their entirety. Due to the indices on identifier attributes, this can be avoided. Actually, the join on identifier attributes (marked with a star) is very effective.

Finally, the query of Example 6 (doctors who can treat all diseases of Mr. Brown) becomes:

\[
\text{TEMPVII}_i := \pi_{\text{identifier}}(\sigma_{\text{name}=\text{Brown}} \text{PAT\_ROOT}_i) \quad (1 \leq i \leq N), \\
\text{TEMPVIII}_i := \pi_{\text{disease}}(\text{TEMPVII}_j \supseteq \text{identifier}=\text{tid}(r) \text{PAT\_ILLN}_i) \quad (1 \leq i \leq N, \text{TEMPVII}_j \neq \emptyset), \\
\text{TEMPIX}_i := \pi_{\text{name}, \text{disease}}(\text{DOC\_ROOT}_i \supseteq \text{identifier}=\text{tid}(r) \bigcup_{j=1}^{N} \text{DOC\_DISE}_j) \quad (1 \leq i \leq N), \\
\text{RESULT6} := \bigcup_{i=1}^{N} \left( \text{TEMPIX}_i \div \left( \bigcup_{j=1}^{N} \text{TEMPVIII}_j \right) \right) \tag{6}
\]

Again, we assume there is a single patient called Mr. Brown. TEMPVII will contain his identifier. TEMPVIII is the set of identifiers of all his illnesses. TEMPIX contains the name and specializations of all doctors. Since each doctor has only a single root tuple, it can be generated in parallel. After collecting/replicating TEMPVIII on all nodes (TEMPIX fragments remain where they have been generated), the result is generated by a divide.

4.1.3 Normalized storage model with intra tuple-object clustering (t-NSM)

The formulas for t-NSM are almost equal to those for f-NSM. However, since all data of a single tuple-object has been stored on the same node, less data needs to be replicated (or redistributed) over all nodes. The result of Example 1 (names and symptoms of all illnesses) will be built like:
RESULT1 := ∪_{i=1}^{N} (\pi_{\text{name}, \text{name}, \text{description}} (\text{ILL\_ROOT}_{i} \bowtie^*_{\text{identifier}=\text{tid}(r)} \text{ILL\_SYMP}_{i})) \quad (7)

Due to the intra tuple-object clustering, ILL\_ROOT and ILL\_SYMP can be joined without redistributing them first.

The query of Example 2 (doctors working in the hospital 'central') becomes:

\[
\begin{align*}
\text{TEMPI}_i & := \pi_{\text{doctor}}((\sigma_{\text{name}=\text{central}} \text{HOS\_ROOT}_{i} \bowtie^*_{\text{identifier}=\text{tid}(r)} \text{HOS\_DOCT}_{i}) , \\
\text{TEMPII}_i & := \text{TEMPI}_i \bowtie^*_{\text{doctor}=\text{identifier}} \text{DOC\_ROOT}, \quad (1 \leq i \leq N, \text{TEMPI}_i \neq \emptyset), \\
\text{TEMPIII}_i & := \text{TEMPI}_i \bowtie^*_{\text{doctor}=\text{tid}(r)} \text{DOC\_ADDR}, \quad (1 \leq i \leq N, \text{TEMPI}_i \neq \emptyset), \\
\text{TEMPIV}_i & := \text{TEMPI}_i \bowtie^*_{\text{doctor}=\text{tid}(r)} \text{DOC\_PHON}, \quad (1 \leq i \leq N, \text{TEMPI}_i \neq \emptyset), \\
\text{TEMPV}_i & := \text{TEMPI}_i \bowtie^*_{\text{doctor}=\text{tid}(r)} \text{DOC\_DISE}, \quad (1 \leq i \leq N, \text{TEMPI}_i \neq \emptyset), \\
\text{RESULT2} & := \bigcup_{i=1}^{N} (\text{TEMPII}_i, \ldots, \text{TEMPV}_i) \quad (8)
\end{align*}
\]

The relevant doctor identifiers (TEMPI) are generated on a single node. After distributing TEMPI over all nodes, the doctor information can be retrieved by all nodes in parallel (TEMPII–V), as well as the result (RESULT 2) can be constructed (joins) by all nodes in parallel without any further communication. A union is only needed to gather all result tuple-objects.

Finally, the query of Example 6 (doctors who can treat all diseases of Mr. Brown) becomes:

\[
\begin{align*}
\text{TEMPVI}_i & := \pi_{\text{disease}}((\sigma_{\text{name}=\text{Brown}} \text{PAT\_ROOT}_{i} \bowtie^*_{\text{identifier}=\text{tid}(r)} \text{PAT\_ILLN}_{i}) , \\
\text{TEMPVII}_i & := \pi_{\text{name}, \text{disease}} (\text{DOC\_ROOT}_{i} \bowtie^*_{\text{identifier}=\text{tid}(r)} \text{DOC\_DISE}_{i}), \quad (1 \leq i \leq N), \\
\text{RESULT6} & := \bigcup_{i=1}^{N} (\text{TEMPVII}_i + \text{TEMPVI}_i) \quad (\text{TEMPVI}_i \neq \emptyset) \quad (9)
\end{align*}
\]

The set of identifiers of Mr. Brown's illnesses (TEMPVI) is generated on a single node. At the same time, all nodes generate the relevant doctor information of the doctors they 'own' (TEMPVII). After TEMPVI has been replicated on all nodes (in its entirety), the result is generated in parallel by a divide.

4.2. How to compute response times

This section tells how to calculate the response times for the different tuple-object operators (thus how to translate our formulas into times). First we will treat some general issues. Then we will focus on disk I/O, network messages, and CPU load in separate subsections.

4.2.1 General rules

First of all, we think it is wise to work with average values. That is, we base our computation on reals rather than integers, and we ignore ceilings. For example, if X is the
average number of pages a relation has been stored on (not necessarily an integer), then each of the N nodes owns X/N pages, etc. Not taking average values is unnecessarily detailed while not giving more insights. Moreover, since not all tuple-objects of a single type are as large, tuple-object parameters (such as their size) tend to be average values.

While constructing our formulas, we assumed that it was known which tuple-objects had been stored on which nodes (DSM, t-NSM) or on which node a tuple-object's root tuple had been stored (t-NSM, t-NSM). We assume that a hash function applied on the tuple-object identifier gives this node information.

Given their formulas, the queries will be evaluated as smart as possible. Selections, projections and dot operations are combined to a single action in order to save CPU and IO time. Also, 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. Instead of saving the incoming tuples for further processing, they are used before all input is available. Pipelining is used even when a join (to reconstruct doctors) and a division (to construct result) are executed behind each other (Example 6, NSM).

To be able to determine execution times we need some parameter settings. Table 1 shows the system parameters with their settings. Most settings have been based on values as presented in the literature [25]. From the description as presented in the table, all parameters should be self-explanatory.

Finally, we also need some tuple-object parameters. Suppose a tuple-object of type Doctor contains (on the average) 2 phone numbers and 100 references to a disease. Also, a tuple-object of type Patient has (on the average) 5 elements in the set attribute illness and a tuple-object of type Illness has 5 elements in the set attribute symptom. A tuple-object of type Hospital has (again on the average) 25 references to a doctor and 500 references to a patient. With a string attribute being 40 byte and all other attributes (including identifier attributes) being 8 byte, Table 2 shows the number of 'tuples' and the tuple size for all example (nested) relations. We use the following notation. If R is a (nested) relation, ||R|| is the number of tuple(-object)s in R, tuple_size(R) is the size of a tuple in R and IRRI is the number of pages used to store R on. Obviously |R| = ||R|| * tuple_size(R) * Fpgoc/PG (N.B., for Fpgoc and PG: see Table 1).

We will compute and present the I/O, network and CPU times independently of each

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>F_{hash}</td>
<td>space overhead factor for hashing (join)</td>
<td>1.2</td>
</tr>
<tr>
<td>F_{pgoc}</td>
<td>number of pages to store PG byte on (= 100%/degree of page occupation)</td>
<td>1.4 (=100%/70%)</td>
</tr>
<tr>
<td>M</td>
<td>main memory size per node for data</td>
<td>512 pages (=1 Mbyte)6</td>
</tr>
<tr>
<td>N</td>
<td>number of nodes in system</td>
<td>10 nodes</td>
</tr>
<tr>
<td>PG</td>
<td>size of a memory/disk page</td>
<td>2048 byte</td>
</tr>
<tr>
<td>sel</td>
<td>query selectivity (divide operation)</td>
<td>0.01</td>
</tr>
<tr>
<td>t_{comp}</td>
<td>CPU time to compare two values</td>
<td>0.2 \mu s</td>
</tr>
<tr>
<td>t_{hash}</td>
<td>CPU time to hash a value</td>
<td>0.9 \mu s</td>
</tr>
<tr>
<td>t_{move}</td>
<td>CPU time to move data</td>
<td>0.1 \mu s/byte</td>
</tr>
<tr>
<td>t_{IO}</td>
<td>time to read/write a disk page</td>
<td>20 ms/page</td>
</tr>
<tr>
<td>t_{MSG}</td>
<td>average time to send a network message</td>
<td>1.5 \mu s/byte6</td>
</tr>
</tbody>
</table>

5This value seems ridiculous small. However, our example tuple-objects are small as well (in size and number).
6Obviously 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.
Table 2
Parameter settings for the different (nested) relations

<table>
<thead>
<tr>
<th>Nested relation</th>
<th>Number of tuples</th>
<th>Tuple size (byte)</th>
<th>Number of pages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$</td>
<td>R</td>
<td>\</td>
</tr>
<tr>
<td>DSM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOCTOR</td>
<td>2500</td>
<td>952</td>
<td>1.63 * 10^2</td>
</tr>
<tr>
<td>PATIENT</td>
<td>50000</td>
<td>304</td>
<td>1.04 * 10^4</td>
</tr>
<tr>
<td>ILLNESS</td>
<td>10000</td>
<td>488</td>
<td>3.34 * 10^3</td>
</tr>
<tr>
<td>HOSPITAL</td>
<td>100</td>
<td>4288</td>
<td>2.93 * 10^2</td>
</tr>
<tr>
<td>NSM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOC_ROOT</td>
<td>2500</td>
<td>48</td>
<td>8.20 * 10^1</td>
</tr>
<tr>
<td>DOC_DISE</td>
<td>250000</td>
<td>16</td>
<td>2.73 * 10^3</td>
</tr>
<tr>
<td>DOC_ADDR</td>
<td>2500</td>
<td>96</td>
<td>1.64 * 10^2</td>
</tr>
<tr>
<td>DOC_PHON</td>
<td>5000</td>
<td>16</td>
<td>5.47 * 10^1</td>
</tr>
<tr>
<td>PAT_ROOT</td>
<td>50000</td>
<td>48</td>
<td>1.64 * 10^3</td>
</tr>
<tr>
<td>PAT_ILLN</td>
<td>250000</td>
<td>32</td>
<td>5.47 * 10^3</td>
</tr>
<tr>
<td>PAT_ADDR</td>
<td>50000</td>
<td>96</td>
<td>3.28 * 10^3</td>
</tr>
<tr>
<td>PAT_PRIV</td>
<td>50000</td>
<td>56</td>
<td>1.91 * 10^3</td>
</tr>
<tr>
<td>ILL_ROOT</td>
<td>10000</td>
<td>88</td>
<td>6.02 * 10^2</td>
</tr>
<tr>
<td>ILL_SYMP</td>
<td>50000</td>
<td>88</td>
<td>3.01 * 10^3</td>
</tr>
<tr>
<td>HOS_ROOT</td>
<td>100</td>
<td>88</td>
<td>6.02 * 10^2</td>
</tr>
<tr>
<td>HOS_DOCT</td>
<td>2500</td>
<td>16</td>
<td>2.73 * 10^1</td>
</tr>
<tr>
<td>HOS_PATA</td>
<td>50000</td>
<td>16</td>
<td>5.47 * 10^2</td>
</tr>
</tbody>
</table>

other. The advantage of such an approach is that it facilitates the conversion (adaptation) of our conclusions to other systems or new technologies. For example, if someone has a database system which resembles our system except for the network, which is a wide area network ($t_{MSG} = 0.02 \text{ms/byte}$) or a fast local area network ($t_{MSG} = 0.1 \mu\text{s/byte}$), then our conclusions will hold for this system as well, provided the importance of network times is increased or decreased respectively. Or, if new CPUs that are ten times as fast become available, the conclusions can be easily adapted by lowering the importance of CPU time as compared with disk I/O and message times.

We will present (section 5) disk I/O, message and processor (CPU) times. We emphasize that the processor times we present do not include times related to disk I/O or network messages. CPU costs due to disk I/O and message passing will be added to the disk I/O and message (rather than the processor) costs respectively. In general, these indirect CPU costs may be considerable. For instance, the costs of sending a message are actually almost entirely determined by software overhead and not transmission time.

While adding the disk I/O, message and processor times to a single response time, we neglect a possible overlap. This seems reasonable since it appears that disk operations have to be executed first and message operations serve as a synchronization mechanism, while both kinds of operations require the CPU (which is not the bottleneck).

4.2.2 Disk I/O
Indices On each node, local indices exist that relate the identifiers of the tuple-objects, which this node stores, to pages. Suppose DSM has a single index entry per tuple-object instance. Assuming each entry is 12 byte (an 8 byte identifier value and a 4 byte local pointer), and using our default values (Table 2), we need 73.4 Kbyte for indices per node. With t-NSM, there is a single index entry for each tuple-object instance that is owned by a node as well. However, since a tuple-object is stored distributed over many relations, each index entry is 8 byte (identifier value), plus the number of relations a tuple-object has been stored over times 4 byte (pointers). Notice that the length of an index entry varies per
Joining distributed complex objects

tuple-object type. For our example tuple-objects, this makes 139 Kbyte per node, for all indices in total. With f-NSM, the indices are identical to those for t-NSM. But, since each tuple-object is distributed over all nodes, the index space per node is N times as large as compared with t-NSM (N is the number of nodes). As we have 10 nodes in our example, with f-NSM the indices take 1.39 Mbyte per node.

Such an index becomes unacceptably large, in particular for f-NSM. An alternative is to have one index entry per page. If a tuple-object is larger in size than a page there will be several index entries per tuple-object. However, when several tuple-objects fit on a single page the index will be non-dense, that is there are less index entries than tuple-objects. The latter situation, which reduces the index size, is likely to happen with NSM where tuple-objects are distributed over several relations and possibly several nodes. Suppose all index entries are 12 byte, 4 byte for the page pointer and 8 byte for the identifier of the (first) tuple-object stored on it. Using the number of pages as presented in Table 2, for our example tuple-objects all indices together take 18.4 Kbyte/node with DSM, and 22.9 Kbyte/node with either f-NSM or t-NSM. Such small indices can easily be kept in main memory. Notice that the use of a non-dense index assumes that these tuple-object identifiers can be ordered. For, there will be no index entry for each tuple-object identifier and still each tuple-object identifier has to be related to a page number. Also, we have to know which indices belong to which tuple-object types/relations.

There is yet another possibility. The identifier attributes may contain some page information (they might even be physical pointers). For instance, we assume all identifier and tid attributes to be 8 byte. A tuple-object identifier (and thus the root part of a tid) might be 4 byte. The remaining space (4 byte) in the identifier attribute can be used for additional information, such as information with regard to access permission, locking, physical page address, etc. In a tid attribute, the parent and own part might be small local pointers (2 byte) containing some physical page information.

Concluding, we will neglect index accesses. That is, we assume a tuple-object identifier can be transformed to a physical page address for a specific (nested) relation without requiring costs to be considered. Therefore, given a (tuple-object) identifier, a minimum number of page accesses will be needed.

**Page fetches** A very important point with regard to disk I/O is that 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 \(|R|/N\)*t\(_{IO}\). In general, the time to read a local (nested) relation from disk is equal to the number of pages to fetch times t\(_{IO}\).

If a number of tuple(-object)s of the same type have been randomly distributed over the pp pages a certain file consists of, possibly not all pp pages need to be fetched from disk if only a fraction of these tuple(-object)s is needed. Suppose tf tuple(-object)s have to be fetched, given their identifiers (which are pointers that somehow provide physical page information). Bernstein [4] derived a formula for the number of pages page_fetches(pp, tf) to input:

\[
\begin{align*}
\text{page-fetches}(pp, tf) = (tf + pp)/3 & \quad \text{if } pp/2 < tf < 2*pp \\
pp & \quad \text{if } 2*pp < tf \\
\end{align*}
\]

This 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 pages(pp, tf, ts) if tf tuple(-object)s, which are ts byte in size and have been randomly distributed over pp pages, have to be fetched from disk.
This formula may be explained as follows. If a tuple(-object) is smaller in size than a page, we assume a tuple(-object) does not span pages and use Bernstein's formula. Obviously, with $ts < pg$ the floor function returns a zero and equation 11 actually is equal to equation 10. The unit of size is an entire byte, so 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 (this might be achieved by, e.g. storing each tuple starting on a new empty page). Thus, a tuple whose size is 1.5 times the page size will be stored on two, and never on three pages. Since a tuple(-object) is stored contiguously on disk, $[(ts - 0.5)/pg]$ 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. This might seem strange at first, but it gives a reasonable approximation. With a few tuple(-objects) to input, $[(ts - 0.5)/pg]$ pages per tuple(-object) will be fetched. With many tuple(-objects) to input, all pages will be fetched. In an in-between situation, finally, the parts of two successive tuple-objects that are still to fetch may not be on the same page.

If the tuples are clustered rather than randomly distributed over the pages, we can see each cluster as a single unit which has to be fetched. Suppose $tf$ tuple(-object)s have to be fetched. They have been clustered in groups of $cs$ tuple(-object)s and 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. The number of page accesses $pages'(pp, tf, ts, cs)$ is:

$$pages'(pp, tf, ts, cs) = pages(pp, \frac{tf}{cs}, ts*cs)$$

### 4.2.3 Network messages

We assume the communication to be 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 $t_{msg}$. In the formulas of section 4.1, message passing is (implicitly) indicated by a union or by a relation fragment $TEMP_i$ that is generated on a single node and used on all nodes. If a (nested) relation $R$ has to be sent over the network (e.g. when the results are sent to the output), the corresponding network message time is $||R|| * tuple_size(R) * t_{msg}$. We will examine some of the often occurring situations more closely.

Suppose relation $R$ has been distributed over all nodes, each node having a fragment $R_i$ $(1 \leq i \leq N)$. If the entire relation $R$ is needed on all nodes, data messages will be needed to get a union of all fragments on all nodes. Two common situations with their communication costs:

(a) only fragment $R_j$ is non-empty, the costs: $||R_j|| * tuple_size(R) * (N - 1) * t_{msg}$,

(b) all fragments are non-empty, the costs: $N * (||R||/N) * tuple_size(R) * (N - 1) * t_{msg}$.

If the relation has to be (re)distributed over (rather than replicated on) all nodes, each fragment tuple is only sent to a single node (rather than all nodes), with a hash function determining which node to send a tuple to:

(c) only fragment $R_j$ is non-empty: $||R|| * (t_{hash} + tuple_size(R) * (t_{move} + (N - 1)/N + t_{msg}))$,

(d) all fragments are non-empty: $(||R||/N) * (t_{hash} + tuple_size(R) * (t_{move} + N * (N - 1)/N + t_{msg}))$,
Each node has to send a fraction \((N - 1)/N\) of its local tuples to other nodes. In case (c), however, only node \(j\) needs to send data since the fragments of the other nodes are empty. Some processor time is involved as well. A hash function determines which node to send a tuple to whereupon the tuple is moved to an output buffer. In case (d), this processor load is executed in parallel by all nodes. Notice that possibly not all processor costs have been presented. With NSM, the tuples of a relation are clustered on the \(t\_id\) attribute. Therefore, after the distribution step each node has to merge the relation fragments it received. On each node, \(N\) fragments of size \(\|R\|/N^2\) are merged. The costs: \((\|R\|/N)(t_{hash} + \text{tuple\_size}(R)\)*t\_move). Possibly, the arising relation fragment has to be temporarily stored on disk as well!

4.2.4 Processor (CPU) time

In this subsection we concentrate on the processor time needed for specific tuple-object operations. We will not show the costs for all operations. However, the costs for the operations that are not shown should be trivial after reading this section. Of course, though not presented in this subsection, each operation might involve disk I/O and message passing costs as well.

**Projection** A projection \(\pi_X R\) scans the tuples of \(R\) one by one, moving the right attributes to a result tuple. The processor costs are \(\|R\| \ast \text{tuple\_size}(\pi_X R)\)*t\_move.

**Selection** Identically, a selection \(\sigma_F R\) scans all tuples, tests the selection predicate and moves matching tuples to a result tuple. The processor costs are \(\|R\| \ast t_{comp}\) to test all tuples and \(\|\sigma_F R\| \ast \text{tuple\_size}(R)\)*t\_move to construct the result.

Notice that, due to combining operations and pipelining, the cost for a sequence of two operations may be smaller than the sum of the costs for these two individual operations. For example, the processor costs for a selection on \(R\) followed by a projection \((\pi_X \sigma_F R)\) will be \(\|R\| \ast t_{comp} + \|\sigma_F R\| \ast \text{tuple\_size}(\pi_X R)\)*t\_move.

The costs for a selection will be reduced if an index is used. Due to the index, less tuples will be fetched from disk and therefore less tuples need to be scanned.

**Join** We assume a hash-based join algorithm for the join operation \(R \bowtie S\). Suppose \(R\) is the inner (smaller) relation. First a hash table is built for the inner relation, the processor costs: \(\|R\| \ast (t_{hash} + \text{tuple\_size}(R)\)*t\_move). Then the outer relation is probed against this hash table: \(\|S\| \ast (t_{hash} + t_{hash} \ast t_{comp})\). Finally a result tuple is constructed for each pair of matching join attributes: \(\|R\| \ast \|S\| \ast \text{sel}\ast (\text{tuple\_size}(R) + \text{tuple\_size}(S))\)*t\_move. The parameter ‘sel’ indicates the join selectivity. With a semijoin \(R \bowtie S\) the result is constructed from \(S\) attributes only. Consequently \((\text{tuple\_size}(R) + \text{tuple\_size}(S))\) becomes \(\text{tuple\_size}(S)\) in the result construction phase.

If the (local) 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 there is enough main memory space available. One by one these smaller joins are executed. These buckets will be temporarily stored on disk. So the total costs for a bucket forming phase will be \(\|R\| \ast (t_{hash} + \text{tuple\_size}(R)\)*t\_move) + \|S\| \ast (t_{hash} + \text{tuple\_size}(S)\)*t\_move) + (\|R\| + \|S\|)*2\*t_{io}. If a relation has been stored on disk, sorted on the identifier attribute (on which an index exists), a bucket forming phase can be omitted if the identifier is the join attribute. For, a partition based on the join attribute already exists.

The reconstruction of a tuple-object from its normalized representation involves a number of joins on identifier attributes. All operand relations have been clustered/sorted on
identifier. We assume that a hash table is made for the root tuples, that all remaining participating relations are probed against this hash table\(^7\), and that on a match the relevant data is moved to an in-memory result space where the tuple-objects are constructed. In particular the latter point differs from a normal join since not the entire result is constructed in a single pass. So numerous disk I/Os are prevented by implementing the series of joins in a clever way.

**Division** In the join query of Example 6, the division operation \(R \div S\) shows up. In this example, \(S\) is usually small (fits in the available main memory) and \(R\) has been sorted (or rather clustered) on the first attribute. Under these conditions the processor costs resemble those for a join. First a hash table will be made for \(S\): 
\[
\|S\| \ast (t_{hash} + \text{tuple} \_\text{size}(S) \ast t_{move})
\]
Then \(R\) is probed against the hash table: 
\[
\|R\| \ast (2 \ast t_{comp} + t_{hash} + F_{hash} \ast t_{comp})
\]
Actually this is worst case since all tuples of \(R\) are probed against the hash table. Compared with the join, an extra \(2 \ast t_{comp}\) is included with this probing phase in order to both test the first attribute of \(R\) to have still the same value, and to increment/test a counter. When \(\|S\|\) tuples (test counter!) of \(R\) with identical value for the first attribute have matched the join table, a result tuple is constructed: 
\[
\|R\| \ast \text{sel} \ast \text{tuple} \_\text{size}(\pi R) \ast t_{move}
\]

5. Results

**Table 3** shows the response times for the three example join queries if our default parameter values are used. Times are presented for all three storage structures: the direct storage model (DSM), the normalized storage model without (f-NSM), and with (t-NSM) intra tuple-object clustering. We will analyze these results query by query.

5.1. Response times for example 1

For the query of Example 1, the selection of all \textit{I11ness} names and symptoms, the disk I/O time for all three storage structures is about equal (Fig. 14). In all cases all data of all tuple-objects of the type \textit{I11ness} has to be retrieved from disk. With NSM the \textit{I11ness} table

<table>
<thead>
<tr>
<th>TABLE 3</th>
<th>Calculated default response times (s) for the example join queries</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DISK I/O + MESSAGES + PROCESSOR = TOTAL</td>
</tr>
<tr>
<td>DSM</td>
<td>6.67 + 6.60 + 0.0440 = 13.3</td>
</tr>
<tr>
<td>f-NSM</td>
<td>7.22 + 14.9 + 0.172 = 22.3</td>
</tr>
<tr>
<td>t-NSM</td>
<td>7.22 + 9.00 + 0.0754 = 16.3</td>
</tr>
<tr>
<td>Example 2 (a functional join)</td>
<td></td>
</tr>
<tr>
<td>DSM</td>
<td>0.636 + 0.0360 + 0.000311 = 0.673</td>
</tr>
<tr>
<td>f-NSM</td>
<td>0.741 + 5.87 + 0.0993 = 6.71</td>
</tr>
<tr>
<td>t-NSM</td>
<td>0.232 + 0.0360 + 0.0303 = 0.298</td>
</tr>
<tr>
<td>Example 6 (a value-based join)</td>
<td></td>
</tr>
<tr>
<td>DSM</td>
<td>24.0 + 0.00204 + 0.0605 = 24.1</td>
</tr>
<tr>
<td>f-NSM</td>
<td>8.93 + 5.40 + 0.252 = 14.6</td>
</tr>
<tr>
<td>t-NSM</td>
<td>8.93 + 0.00204 + 0.189 = 9.13</td>
</tr>
</tbody>
</table>

\(^7\)To minimize main memory needs, these participating relations are probed simultaneously rather than one by one. In this way the result tuple-objects are (re)constructed one by one (as far as possible), rather than all at the same time.
Joining distributed complex objects

Fig. 14. I/O, message and CPU times for the query of Example 1 ('materialized' join).

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

The network message cost are minimal with DSM as well. The only communication needed is for sending the result to the output. This holds true for t-NSM as well, but the result is larger since it is in a relational (rather than direct) representation. With f-NSM, however, a lot more communication is needed since in general the illness' name and symptoms have initially not been stored on the same node. Notice the different vertical scales in Fig. 14!

Like the message cost, the processor costs are the least for DSM and the most 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) and, since there is no intra tuple-object clustering but the data has to be on a single node before it can be joined, processor time is involved with redistributing relations.

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

This effect can be seen from Fig. 15 as well, where we varied the number of nodes in the system. With f-NSM, the relation ILL_SYMP is redistributed over the nodes in order to get all tuples belonging to the same Illness tuple-object on the same node. Each node merges the small sub-fragments of ILL_SYMP, which it owns after redistribution, into a single new ILL_SYMP fragment (that obeys the NSM demands). If the total number of nodes becomes smaller, each node will own more tuples of ILL_SYMP, and possibly the ILL_SYMP fragment can not be kept in main memory after redistribution. Figure 15 clearly shows that this happens if the number of nodes becomes less than eight. Although not visible from the figure, it appears that a hash table overflow does still not occur with this example.

Figure 15 also shows that for all storage models the total response time \( t_{IO} + t_{MSG} + t_{CPU} \) decreases with increasing number of nodes. This is due to the fact that the disk I/O and processor load is distributed over more nodes. If \( N \rightarrow \infty \) (or actually if \( N \rightarrow 10000 \) because...
there are only 10000 tuple-objects of the type Illness) the total response time approaches the communication time $t_{msg}$, which is constant for DSM and t-NSM and which slightly increases (with an increasing number of nodes) for f-NSM. This slight increase in communication time with f-NSM is due to the fact that the data is distributed over nodes, which makes the probability that a tuple has initially already 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

For the query of Example 2, select all doctors working in the hospital ‘central’, the disk I/O time is minimal for t-NSM (Fig. 16). 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.

Fig. 15. Total response time while varying the number of nodes with the query of Example 1.

Fig. 16. I/O, message and CPU times for the query of Example 2 (‘functional’ join).
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 and, as far as possible, only for this single selected tuple-object the remaining data is fetched from disk. However, for the second part of the query, which involves 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 disk I/O since (per node) the input is from a single file (this is not shown in the figure). The reason why t-NSM needs less disk I/Os as compared to f-NSM is that with t-NSM all tuples belonging to a single doctor are stored on a single node, clustered together per relation. With f-NSM these tuples are distributed over the nodes, and per node clustered together in the corresponding relations. 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. This fact even makes the disk I/O time of f-NSM the worst of all.

The communication time is worst for f-NSM as well. It almost entirely determines the total response time for this storage structure. Since the result tuple-objects of the type Doctor are constructed in parallel, all the information of a single doctor needs to be collected on a single node, causing many network messages. On the contrary, the communication costs for the other storage models is almost negligible. Since there is 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 slightly increases with an increasing number of nodes, provided that the number of nodes is large enough to prevent main memory overflow. With a small number of nodes (less than six) main memory overflow occurs in the f-NSM case, disk I/O becomes important and the total response time decreases with an increasing number of nodes.

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

While 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). With 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 have not been equally distributed over the nodes) or only a few number of tuple-objects to be fetched (there are more nodes than tuple-objects to be fetched from disk). In both cases f-NSM has the advantage over DSM and t-NSM to enable much more parallel disk I/Os [26].

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, though it is fast 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 6

For the query of Example 6, what are the names of the doctors who can treat all Mr. Brown's diseases, the disk I/O time is minimal for the normalized storage models (Fig. 17). 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. Entire tuple-objects of this type are
retrieved whereas we only want to test the name attribute to be Mr. Brown. 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 know 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, for other (application specific) value-based joins we expect the same trends for the results.

Varying the number of nodes in the system (Fig. 18) 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 with the number of nodes smaller than six since the relation DOC_DISE is temporarily stored on disk after redistribution. However, even for this situation DSM still has the largest response time.

Fig. 17. I/O, message and CPU times for the query of example 6 ('value-based' join).

Fig. 18. Total response time while varying the number of nodes with the query of Example 6.
With regard to other parameter settings, we wonder what happens if multiple tuple-object attributes are involved in the join condition. In particular this is the case if (due to the join predicate) non-atomic tuple-object attributes have to be compared. Since with non-atomic attributes recursion may turn up (if, e.g., a relation attribute is a relation again), many tuple-object attributes may be involved in testing the join condition. The more join attributes we have the more profitable it is to use a direct storage model with regard to disk I/O. For with a normalized model, unless the join attributes are part of the same sub-relation, for each join attribute a separate relation has to be fetched from disk to get all join attributes in main memory.

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 from too much communication. A direct storage model suffers from the necessity to input entire tuple-objects.

5.4. **Summary of the results**

The results may be summarized as follows. With DSM, the retrieval of an entire tuple-object is very fast. Also, no joins are needed to reconstruct tuple-objects and, since objects are stored on a single node, the network communication is minimal. Therefore, DSM is very well suited for materialized joins. DSM may be good in functional joins as well, but suffers from the fact that 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. Much superfluous data will be retrieved.

Since a normalized storage model is much more effective in retrieving only those data that are actually needed, it might much better support functional joins. But, some points have to be taken care of. With NSM, many joins will be needed to reconstruct the tuple-objects from their normalized representations. Although tuple identifiers seem to support such a reconstruction very well, these joins are still not for free. Very important is the effect of buffer size. If the main memory is too small, the larger join has to be split into a number of smaller joins. Such a step is very expensive because it requires many additional disk I/Os and processor operations. Also, performance problems show up if all data belonging to a single tuple-object is not stored on a single node (f-NSM). Then, the sooner or later the data has to be collected on a single node. First, the communication overhead of this data redistribution step is unacceptably large. Second, depending on the buffer size, possibly the data has to be temporarily stored on disk after redistribution. Expensive disk I/Os will be involved.

With value-based joins, the operations that are specific for this kind of join in particular have a comparable performance for all three storage structures. Therefore, the conclusions resemble those for a functional join. The best support is provided by t-NSM. However, with many join attributes, DSM has the advantage of retrieving all attributes at once.

Finally, DSM and t-NSM are the best with regard to scaling the system. The disk I/Os and processor load are almost equally distributed over all nodes. With f-NSM the number of nodes must not be too large because the communication overhead will increase with the number of nodes. However, the number of nodes must not be too small as well, because then the local buffer may be too small for all data a node receives after the redistribution of a relation.

6. **Conclusions**

In this paper we described hierarchically structured complex objects, which we called tuple-objects. We showed that three kinds of tuple-object joins may be distinguished. A
materialized join is simply the projection on a nested attribute. A functional join is the navigation along object references. Finally, a value-based join is based on the comparison of attribute values, but, as opposed to the relational join, these attribute values can be non-atomic. For each kind of join, we thoroughly investigated an example query. We analytically analyzed the disk I/O, network message, and processor costs involved with the queries. We did so for three different complex object storage structures.

The results show that materialized joins are best supported by a direct storage model for tuple-objects (DSM). A functional join may be well supported by a direct storage model as well, but a normalized storage model with intra tuple-object clustering (t-NSM) is generally better. It allows a strategy in which first those parts of a tuple-object are retrieved, which might cancel the retrieval of the other parts. Joins to reconstruct the tuple-object from its normalized representation will be supported by identifier attributes. Not clustering tuple-objects on a single node (f-NSM), however, is rarely a good idea because of the communication overhead. For a value-based join identical conclusions hold. In general, t-NSM seems to be the best overall storage structure for tuple-objects, but, of course, this depends on the system and workload.

The workload of the system shows us the execution frequencies of all queries. Therefore, it tells us how often each kind of join query is executed. The workload depends on the application. Although for an average application t-NSM might be the best, t-NSM will not always be the best. In particular, in non-standard database environments the materialized joins seem to be the most important whereas the traditional value-based joins are not very important. Therefore, DSM might be the best storage structure as well. Notice that within a certain class of join queries, individual joins may prefer different storage structures as well. For instance, if there is a single small join attribute in a large tuple-object, t-NSM will be chosen for its flexible disk I/Os. With many large join attributes, however, DSM will be used since it inputs all (join) attributes with a minimal number of disk accesses.

The system characteristics are important as well. We showed the results for disk I/O, network messages, and processor costs independent of each other, which 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 these costs factors and might change the conclusions correspondingly. For instance, since the communication and processor costs are minimal with DSM, DSM should be used if network messages and CPU instructions are expensive. If the disk I/O costs per byte are very high, NSM should be used to avoid the input of data that are not needed. Since the units of retrieval decrease in size from DSM to t-NSM to f-NSM, DSM might be favoured by a large, and f-NSM by a small page size. Also, we should never use f-NSM if the communication is expensive. Finally, factors like the buffer size may be decisive as well. A small buffer makes NSM, and in particular f-NSM, less attractive. If relations do not fit in main memory, the reconstruction joins become much more expensive. Also, possibly the relations have to be temporarily stored on disk after redistribution.

Our results show that, for our parameter settings, CPU costs seem to be unimportant whereas the disk I/O costs are a bottleneck. Therefore, the conclusions based on our default parameter values may be interesting for many systems since in the parallel database world there is a general consensus that CPU costs are negligible, and that the I/O bandwidth is a severe bottleneck.

\*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.

\*Notice that in our results the CPU costs due to message passing have been added to the message (rather than processor) costs. In general these CPU costs are important.
We investigated storage structures for tuple-objects with regard to join operations. 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. Current and future research focuses on these aspects. Also, other storage structures, as well as in-between storage structures, do exist and need attention. Using indices on non-identifier attributes might give new insights as well. Putting the results of all these research projects together, we can construct a query optimizer for complex objects, i.e., a program that, given a query on complex objects, chooses the best storage structure to access.

References


Wouter B. Teeuw received his master degree in Technical Physics from the Delft University of Technology in 1989. Afterwards, he joined the University of Twente as a Ph. D. student in Computer Science. His research topics are the design and implementation of a complex object server. In particular, he focuses on performance aspects.

Henk M. Blanken joined, after receiving his master degree in Mathematics, in 1966 Phillips Computer Industry (Apeldoorn, The Netherlands). In 1971 he joined the University of Twente as an associate professor of Computer Science. From this university he received a Ph.D. degree in Computer Science in 1984. During a one-year visit to the IBM Scientific Center at Heidelberg he contributed to the AIM/II project. His current research interests are non-standard database applications and storage structures.