

Duplicate Detection and Deletion in the Extended NF^2 Data Model

K. Küspert, G. Saake*, L. Wegner[§]

IBM Heidelberg Scientific Center, D-6900 Heidelberg, West Germany

*on leave from TU Braunschweig, FB Informatik, D-3300 Braunschweig, West Germany

[§]Gh Kassel - Universität, FB Mathematik, D-3500 Kassel, West Germany

Abstract. A current research topic in the area of relational databases is the design of systems based on the Non First Normal Form (NF^2) data model. One particular development, the so-called extended NF^2 data model, even permits structured values like lists and tuples to be included as attributes in relations. It is thus well suited to represent complex objects for non-standard database applications. A DBMS which uses this model, called the Advanced Information Management Prototype, is currently being implemented at the IBM Heidelberg Scientific Center. In this paper we examine the problem of detecting and deleting duplicates within this data model. Several alternative approaches are evaluated and a new method, based on sorting complex objects, is proposed, which is both time- and space-efficient.

1. Introduction

Within the last decade, the *relational data model* [Co70] has been widely accepted as a basis for commercial database management systems (DBMS's). In the relational model, all data are organized as homogenous collections of tuples within database tables (*relations*). The *tuples* consist of a non-homogenous collection of *attributes*. According to the so-called first normal form (short 1NF) postulate, all attributes must be 'atomic', for instance of type BOOLEAN, INTEGER, REAL, or CHARACTER. The major and well-known benefits of relational database systems are the structural simplicity of the data model combined with a powerful, descriptive query and data manipulation language (DML), e.g. QUEL [St76] or SQL [Ch81].

In the last few years, however, it became increasingly clear that the pure relational (1NF) data model is not very well suited to the management of *complex data objects* in a DBMS. These complex objects occur in many so-called 'advanced' or 'non-standard' application areas, e.g. in engineering (design objects for construction purposes, ...), office automation (structured documents with figures and images, ...), and science (molecule structures in chemistry, ...). If the user wants to retrieve a complex data object as a whole, for instance to display a large engineering design object on the screen or to provide it as input for a construction simulation, a large number of *join* operations are required to (re-)assemble the data object from the corresponding 'flat' tuples and 'flat' tables of a 1NF DBMS. These non-trivial join operations on large amounts of data are very unhandy from the user's point of view, rather error-prone in coding, as well as extremely resource (CPU time and I/O) consuming during execution.

The NF^2 data model [AB84, JS82, Ro85, SS86] is one approach to overcome these limitations of missing 'object orientation' in the 'pure' relational model. This data model ($NF^2 = NFNF =$ Non First Normal Form) combines the 'pure' relational data model and the hierarchical data model, as known, for instance, from IMS [Da81, SJ77]. Attributes in tuples may again be (sub-)relations, i.e. the 1NF postulate is given up deliberately. These concepts of 'nested relations' or 'relations with relation valued attributes' in the NF^2 data model enable the user to map complex objects to *one* NF^2 relation instead of dispersing the data over *several* 'flat' relations.

Finally, the *extended NF^2 data model* [Da86, Pi87] is a further development of the NF^2 data model to capture some additional - for certain applications very important - structural concepts like ordered relations (usually called *lists*), *multisets* (i.e. sets which may have duplicates [Kn73]), nested tuples and

nested (multi)sets, etc. (see Sec. 2 for a more detailed description).

In the last five years, a DBMS prototype, the Advanced Information Management Prototype (AIM-P), based on the extended NF^2 data model, has been developed in a research project at the IBM Heidelberg Scientific Center. At its current stage, AIM-P supports most constructs of the extended NF^2 data model and offers a powerful, high level, SQL like query and data manipulation language, called HDBL (Heidelberg Database Language), for complex object manipulation [PT85, PA86, PT86, ALPS88]. One of the more recent research interests in this project is the provision of suitable mechanisms to *detect and delete duplicates* in the extended NF^2 data model.

In any relational DBMS - be it 1NF, NF^2 or extended NF^2 - the need to detect and delete duplicates arises under two different circumstances:

- 1. Uniqueness on stored database tables:** The user declares a database table as being 'free of duplicates'¹ either for a single attribute or for a collection of attributes. This integrity constraint may be enforced via a primary or secondary key declaration formulated in the system's data definition language (DDL). In the *extended* NF^2 data model, the situation is somewhat more complicated since multisets, nested (multi)sets, etc. may occur where a key declaration in the straightforward sense cannot be done.
- 2. Uniqueness on query results:** The user wishes to obtain a unique *query result*, even if the database tables are not necessarily free of duplicates. This may be the case if e.g. all *different* salaries shall be extracted from a personnel table. In most relational DBMS implementations, the user may add a clause like `UNIQUE` to the query statement (e.g. in SQL) to trigger 'on the fly' duplicate elimination when the query result is built up. The reader should note that according to relational database theory [Co70], a query result must always be a *set* in the mathematical sense, i.e. be free of duplicates. Duplicate elimination, however, is not cheap and most systems therefore trade set theory for performance and do duplicate elimination for query results only on user request (`SELECT UNIQUE ...`).

In most (1NF) relational DBMS products uniqueness on database tables (1) is usually checked and enforced via proper *indexes* (e.g. multiway trees, hash tables); uniqueness on query results (2) is usually achieved via *sorting* accompanied by duplicate elimination. In the case of the NF^2 and *extended* NF^2 data model, however, several new problems arise which have not yet been thoroughly investigated in database research and development:

- What does a term like 'unique' exactly mean for complex objects or parts of complex objects? Do we have to distinguish between different *kinds* of uniqueness, e.g. between 'partial uniqueness', which applies only to the outermost (top) level of a complex object and 'total uniqueness', which implies uniqueness on all levels (cf. [KF88] which recently introduced the terms *shallow* and *deep duplicate free*)?
- Which kind of indexes can be used to enforce uniqueness on (extended) NF^2 database tables?
- Is sorting still the most appropriate method for duplicate elimination 'on the fly' and if yes, how can complex objects, like VLSI circuit layouts, be sorted?

In this paper we primarily concentrate on the topic of duplicate detection and deletion when creating *query results*. We present some basic concepts and algorithms, and discuss a set of design alternatives. In an accompanying paper [SLPW89], DML constructs for requesting uniqueness from the user's point of view are presented in more detail.

1 Throughout this paper the terms 'unique' and 'free of duplicates' are used as synonyms.

2. Data Model and Problem Definition

The extended NF^2 (AIM-P) data model removes many of the unpleasant restrictions on building database structures. Similar to type constructions in programming languages, set-, list- and tuple-constructors can be orthogonally combined to form almost arbitrary object structures. In particular, lists - not supported in the pure NF^2 model - are a commonly used data structure in many applications. Figure 2.1 illustrates the three data models discussed above. Key words in upper case show legal starting points for database object structures.

LISTS	(MULTI)SETS	RELATIONS (SETS)	RELATIONS (SETS)
TUPLES		tuples	tuples
ATOMIC VALUES		atomic values	atomic values
a) AIM-P Data Model (Extended NF^2 Data Model)		b) NF^2 Data Model	c) 1NF Data Model

Figure 2.1: Comparison of three relational data models

Figure 2.2 shows an example for a relation in the extended NF^2 data model. The table gives the available seatings for various types of aircrafts. These types are grouped into three categories (represented as tuples in the database scheme) according to the routes flown. The table itself forms a set as indicated by

{SEATINGS}					
ROUTES	{AIRCRAFT}				
	TYPE	<SEATING_VERSIONS>			
		CODE	[SEATS_IN_CLASS]		
			F	B	T
National	B737	73C1 73C2	8 8	90 102	0 0
	B727	727E	8	131	0
	A310	AB31	18	181	0
Europe & Gulf Region	B727	727E	8	131	0
	A310	AB31	18	181	0
	A300	AB30 AB36	16 18	78 63	72 126
Intercontinental	DC10	DC10	22	76	136
	B747	74M1	21	84	131
		74M2	21	56	177
747E		33	108	204	

Fig. 2.2: Example for a table in the extended NF^2 data model

the set brackets '{' and '}' around SEATINGS. SEATINGS has two attributes, an atomic attribute ROUTES and a non-atomic attribute AIRCRAFT which is again a set, i.e. unordered. For some aircrafts, the same type comes in different versions. The versions are grouped into lists, indicated by '<' and '>' around the attribute SEATING_VERSIONS. The versions are thus ordered, say in ascending sequence according to the lexical ordering of CODE which is the first attribute in SEATING_VERSIONS. The second attribute is SEATS_IN_CLASS, which is of type tuple (indicated by '[' and ']'). Each tuple SEATS_IN_CLASS has three components: F, B, and T for First, Business, and Tourist Class. Here, all three components are of the same type which, in general, does not need to be the case.

The example above contains no multisets. Indeed, if the table had been obtained by projection from the carrier's table of aircrafts in service, a result with possibly a dozen identical entries for each *type* of aircraft would be useless. In other circumstances, however, multisets play a useful role. One example would be a table (set) of medical records which is also made available for statistical research. To protect the individual, all attributes which serve to identify a patient, are hidden by the system for this class of users. Rather than adding an otherwise useless attribute 'NO_OF_MATCHING_INDIVIDUALS' to catch created duplicates by this implicit projection, it would save space and simplify queries to operate with multisets.

A situation where most data models are based on the concept of a set in the mathematical sense, yet implementations of these models usually handle multisets, is rather unsatisfactory. In the extended NF² model, the concepts of set and multiset are well separated. The data manipulation and retrieval language HDBL of the extended NF² data model contains explicit type conversions between the different relation types as well as implicit conventions for the type of query results [PT86].

As an example for duplicate detection we take the (flat) table in Figure 2.3 which relates employees to the files which they may access. The cardinalities of the result tables following projection on FILE_ID should be 4 (i.e. there are 4 distinct files in the table), 3 following projection on NAME, but 4 following projection on (NAME, EMP_NO) because there are two employees with identical NAME (jim) but distinct EMP_NO fields (1024, 9999). Figure 2.4 shows the projection on NAME and EMP_NO, followed by a sort on (NAME, EMP_NO), but preceding duplicate deletion. With the sorted table in Figure 2.4 it is then quite obvious which tuples are duplicates and must be deleted.

Consider now the (extended) NF² data model. The same data as in Figure 2.3 might be arranged in nested format as shown in Figure 2.5. Assume we like to know how many files are distinct in terms of accessing employees. The natural way to proceed is to make a projection on ACCESSEE. Figure 2.6 shows the result before deletion of duplicates. The final result table, however, should have cardinality 3

{ACCESS_RIGHTS}		
NAME	EMP_NO	FILE_ID
susan	3189	customers
jim	1024	trends
jim	9999	budget
susan	3189	budget
jim	1024	customers
susan	3189	trends
jim	1024	accum-sales
peter	2705	budget
jim	9999	trends
susan	3189	accum-sales

Fig. 2.3: A flat table before projection

{ACCESS_TEMP}	
NAME	EMP_NO
jim	1024
jim	1024
jim	1024
jim	9999
jim	9999
peter	2705
susan	3189
susan	3189
susan	3189
susan	3189

Fig. 2.4: ACCESS_RIGHTS projected on NAME and EMP_NO

because both customers and accum-sales are accessed by {[susan, 3189], [jim, 1024]}. Of course, the question is how the DBMS can detect the duplicates. If sorting is the solution, then how can we order complex objects, i.e. is the set {[susan, 3189], [jim, 9999], [jim, 1024]} 'less' than {[jim, 9999], [susan, 3189], [peter, 2705]}?

The problem becomes even more subtle if nested projections are considered. Assume we project in the table from Figure 2.5 on NAME in ACCESSEE. The resulting table, preceding duplicate deletion, is also shown in Figure 2.6. An algorithm which assumes that subobjects are unique might be tempted to use the cardinalities of subobjects as a first criterion to decide on equality. Thus, unless deletion of duplicates is performed from the inside to the outside, {susan, jim, jim} and {susan, jim} might pass as two different objects.

On the other hand, if the result from the projection on NAME is explicitly specified as a multiset, but the projection on the reduced attribute ACCESSEE is supposed to be a set, then the method must distinguish between {susan, jim, jim} and {susan, jim}, yet delete one of the two instances of {susan, jim}. Similarly, if the attribute ACCESSEE were a list, <susan, jim> and <jim, susan> would be two distinct objects.

Another interesting aspect is the detection and deletion of duplicates relative to a subset of the attributes or parts of a value. As an example, a user might consider two entries in a table of scientific abstracts as identical if they agree on 'author(s)' and 'title', disregarding other attributes like 'journal', 'ye-

{ACCESS_RIGHTS}		
FILE_ID	{ACCESSEE}	
	NAME	EMP_NO
trends	susan	3189
	jim	9999
	jim	1024
customers	susan	3189
	jim	1024
budget	jim	9999
	susan	3189
	peter	2705
accum-sales	jim	1024
	susan	3189

Fig. 2.5: ACCESS_RIGHTS in grouped form

{ACCESS_TEMP}	
{ACCESSEE}	
NAME	EMP_NO
susan	3189
jim	9999
jim	1024
susan	3189
jim	1024
jim	9999
susan	3189
peter	2705
jim	1024
susan	3189

{ACCESS_TEMP}
{ACCESSEE}
NAME
susan
jim
jim
susan
jim
jim
susan
peter
jim
susan

Fig. 2.6: ACCESS_RIGHTS projected on ACCESSEE and on NAME

ar_of_publication', 'text_of_abstract'. Furthermore, a DBMS might use a procedure for long text fields whereby two texts are considered equal if they have the same length and a common prefix of, say, 100 characters.

In this context, where two objects are treated as duplicates, but can still be distinguished, the question arises which of the multiple 'identical' objects should be kept: any, always the first in a yet unspecified order, always the last? Moreover, should the 'set' of deleted duplicates be made available to the user? These questions become even more pressing when elimination of duplicates is offered for lists, creating something like a 'unilist' (not offered in the extended NF² data model). Here, the user may even want both the list of unique objects as well as the list of duplicates to retain the relative order of the input sequence. Without an additional hidden attribute, this could not be achieved by sorting alone.

Duplicate detection could also be used as a shortcut for the 'group-by' operator in HDBL. Consider e.g. going from the atomic attribute FILE_ID in ACCESS_RIGHTS to a set valued attribute FILE_IDS. This would group customers and accum-sales into the same tuple because they have matching ACCESSEE's. These questions, however, are outside the scope of this paper and are treated in a discussion of the language interface [SLPW89]. We hope that these examples from the extended NF² data model and the list of problems to solve in detecting and deleting duplicates are sufficient explanation and motivation to look at duplicates in an orderly fashion in the next section.

3. An Ordering Relation for Complex Objects

Any algorithm for duplicate detection and deletion must include two decisions: (1) decide which two objects to compare next and (2) decide whether the two objects are equal. Moreover, we must keep in mind that we are really comparing object *representations*. For objects like (multi)sets, which may have many valid representations with respect to order, one way to answer (2) is to sort the objects internally.

To sort a collection of objects, a linear order relation "<" must be defined on the set *S* of possible objects, s.t. for any three objects *a*, *b*, *c* in *S*, the following conditions are satisfied:

1. Exactly one of $a < b$, $a = b$, $b < a$ is true (law of trichotomy).
2. If $a < b$ and $b < c$, then $a < c$ (law of transitivity).

We may assume that < is defined for atomic objects of the same type (integer, Boolean, real, and character). For texts (which might also be treated as atomic objects), lists, and tuples of atomic objects, the usual lexicographic order can be used, i.e. $A = a_1a_2 \dots a_n < B = b_1b_2 \dots b_m$ iff

1. $n < m$ and $a_i = b_i$ for $i = 1, 2, \dots, n$ or
2. $\exists i \leq m : a_j = b_j$ for $j = 1, 2, \dots, i-1$ but $a_i < b_i$.

Note that for tuples $[a_1, a_2, \dots, a_n]$ we assume a declared order of the columns which needs not necessarily be left to right but must be fixed. The open point is what the ordering is between sets, resp. multisets, e.g. between $\{1, 4\}$ and $\{2, 3\}$.

Let $|S|$ denote the cardinality of (multi)set S . We propose as Definition 1 a recursive, transfinite ordering based on cardinality of (multi)sets and lexicographical ordering, i.e. $\{4\} < \{1, 4\}$ because $|\{4\}| = 1 < 2 = |\{1, 4\}|$ and $\{1, 3\} < \{1, 4\}$ because $|\{1, 3\}| = |\{1, 4\}|$ but $3 < 4$. Applying the order relation recursively to complex objects yields the following definition.

Definition 1 (cardinality-min-ordering):

For objects A and B of the same type, let

$A < B$ iff

- (0) A and B are **atomic** and $A < B$;
- (1) A and B are **(multi)sets** and either $|A| < |B|$ or $|A| = |B|$ and $\text{MIN}(A) < \text{MIN}(B)$ or $|A| = |B|$ and $\text{MIN}(A) = \text{MIN}(B)$ and $A \setminus \text{MIN}(A) < B \setminus \text{MIN}(B)$, where $\text{MIN}(\{e_1, e_2, \dots, e_n\}) = e_i$ with $e_i \leq e_j$ for all $i \neq j$ ($1 \leq i, j \leq n$) and $\{e_1, \dots, e_i, \dots, e_n\} \setminus e_i = \{e_1, \dots, e_{i-1}, e_{i+1}, \dots, e_n\}$, i.e. $X \setminus Y$ denotes X with Y removed;
- (2) A and B are **tuples**, resp. **lists**, and either $\text{FIRST}(A) < \text{FIRST}(B)$ or $\text{FIRST}(A) = \text{FIRST}(B)$ and $A \setminus \text{FIRST}(A) < B \setminus \text{FIRST}(B)$ or A empty and B not empty, where $\text{FIRST}([e_1, e_2, \dots, e_n]) = e_1$, resp. $\text{FIRST}(\langle e_1, e_2, \dots, e_n \rangle) = e_1$. ■

As an example, let us order the elements of the powerset of set $S = \{1, 2, 3\}$. Definition 1 above yields

$$\emptyset < \{1\} < \{2\} < \{3\} < \{1, 2\} < \{1, 3\} < \{2, 3\} < \{1, 2, 3\}.$$

Clearly, this is not the only possible choice for defining the order. For sets we could have used $A < B$ iff $\text{MAX}(A) < \text{MAX}(B)$ or not used the cardinality at all. For lists, on the other hand, we could have included the length as a criterion. Note that tuples of the same type must have the same degree (arity).

Consider what happens if we do not include the cardinality as a first criterion for sets. The resulting order is given in Definition 2.

Definition 2 (no-cardinality-min-ordering):

For objects A and B of the same type, let

$A < B$ iff

- (0) A and B are **atomic** and $A < B$;
- (1) A and B are **(multi)sets** and either $\text{MIN}(A) < \text{MIN}(B)$ or $\text{MIN}(A) = \text{MIN}(B)$ and $A \setminus \text{MIN}(A) < B \setminus \text{MIN}(B)$ or A empty and B not empty, where $\text{MIN}(\{e_1, e_2, \dots, e_n\}) = e_i$ with $e_i \leq e_j$ for all $i \neq j$ ($1 \leq i, j \leq n$) and $\{e_1, \dots, e_i, \dots, e_n\} \setminus e_i = \{e_1, \dots, e_{i-1}, e_{i+1}, \dots, e_n\}$, i.e. $X \setminus Y$ denotes X with Y removed;
- (2) A and B are **tuples**, resp. **lists**, and either $\text{FIRST}(A) < \text{FIRST}(B)$ or $\text{FIRST}(A) = \text{FIRST}(B)$ and $A \setminus \text{FIRST}(A) < B \setminus \text{FIRST}(B)$ or A empty and B not empty,

where $\text{FIRST}([e_1, e_2, \dots, e_n]) = e_1$, resp. $\text{FIRST}(\langle e_1, e_2, \dots, e_n \rangle) = e_1$. ■

As an example we take again the powerset of the set $S = \{1, 2, 3\}$. The resulting sequence is

$$\emptyset < \{1\} < \{1,2\} < \{1,2,3\} < \{1,3\} < \{2\} < \{2,3\} < \{3\}.$$

We note that the *subset relation* (which is not a total ordering) is violated quite frequently and that S itself is "unnaturally" placed in the middle.

If sets are taken from a small ordered universe U , a set S can be represented as a bit-vector V_S (as e.g. in Pascal) where the i 'th bit is 1 in V_S iff the i 'th element in U is an element of the set S . Written from left to right (smallest element in most significant bit position) and interpreted as binary coded integers, the powerset example above yields

$$\begin{aligned} \emptyset < \{3\} < \{2\} < \{2,3\} < \{1\} < \{1,3\} < \{1,2\} < \{1,2,3\} \\ 000 < 001 < 010 < 011 < 100 < 101 < 110 < 111 \end{aligned}$$

which never violates the subset relation, i.e. forms a topological sort for " \subseteq ", but is not "natural" either because $\{3\} < \{1\}$. A fairly natural ordering results from recording the smallest element in the numerically least significant position:

$$\begin{aligned} \emptyset < \{1\} < \{2\} < \{1,2\} < \{3\} < \{1,3\} < \{2,3\} < \{1,2,3\} \\ 000 < 001 < 010 < 011 < 100 < 101 < 110 < 111. \end{aligned}$$

For sets from a potentially infinite universe, we could achieve this ordering using the MAX-function without cardinality, i.e. $\{2, 3\} < \{1, 4\}$ because $\text{MAX}(\{2, 3\}) = 3 < 4 = \text{MAX}(\{1, 4\})$ and $\{1,4\} < \{2, 4\}$ because $\text{MAX}(\{1, 4\} \setminus 4) = 1 < 2 = \text{MAX}(\{2, 4\} \setminus 4)$. This yields Definition 3.

Definition 3 (no-cardinality-max-ordering):

For objects A and B of the same type, let

$$A < B \text{ iff}$$

- (0) A and B are **atomic** and $A < B$;
- (1) A and B are **(multi)sets** and either $\text{MAX}(A) < \text{MAX}(B)$ or $\text{MAX}(A) = \text{MAX}(B)$ and $A \setminus \text{MAX}(A) < B \setminus \text{MAX}(B)$ or A empty and B not empty, where $\text{MAX}(\{e_1, e_2, \dots, e_n\}) = e_i$ with $e_i \geq e_j$ for all $i \neq j$ ($1 \leq i, j \leq n$) and $\{e_1, \dots, e_i, \dots, e_n\} \setminus e_i = \{e_1, \dots, e_{i-1}, e_{i+1}, \dots, e_n\}$, i.e. $X \setminus Y$ denotes X with Y removed;
- (2) A and B are **tuples**, resp. **lists**, and either $\text{FIRST}(A) < \text{FIRST}(B)$ or $\text{FIRST}(A) = \text{FIRST}(B)$ and $A \setminus \text{FIRST}(A) < B \setminus \text{FIRST}(B)$ or A empty and B not empty, where $\text{FIRST}([e_1, e_2, \dots, e_n]) = e_1$, resp. $\text{FIRST}(\langle e_1, e_2, \dots, e_n \rangle) = e_1$. ■

Which definition to choose is largely a matter of taste and convenience. Figure 3.1 shows a listing of book descriptions, where each description can be a subset from {algorithms, data structures, programming}. Definition 1 (the leftmost column using cardinality first, then lexicographical ordering on equal size sets) is quite natural, but so is Definition 2 (middle column, order on repeated minimum element only), because the sets can be read as lexicographically ordered lists. Definition 3 (rightmost column, using the repeated maximum only) seems odd, yet the subset relation is never violated.

As a final example we show the data from Figure 2.6 (subtable on the left) with all (sub)objects brought

cardinality-min	no-cardinality-min	no-cardinality-max
{}	{}	{}
algorithms	algorithms	algorithms
data structures	algorithms data structures	data structures
programming	algorithms data structures programming	algorithms data structures
algorithms data structures	algorithms programming	programming
algorithms programming	data structures	algorithms programming
data structures programming	data structures programming	data structures programming
algorithms data structures programming	programming	algorithms data structures programming

Fig. 3.1: Three orderings of a list of book descriptions

into ascending order according to Definition 1. The result is depicted in Figure 3.2.

The most compelling argument for using either Definition 1 (which we shall use from now on) or Definition 3 (no-cardinality-max-ordering) is the fact that they never violate the subset ordering, i.e. a sorted set of complex objects is always topologically sorted according to the partial subset relation. This even holds for multisets, where multiset X is a subset of multiset Y iff every distinct element in X occurs at least the same number of times in Y as it occurs in X , i.e. $\{a, b\}$ would be a proper subset of $\{a, a, b\}$. For Definition 1 this is trivially seen, for Definition 3 the proof is slightly more complicated, but straightforward, and is omitted here.

{ACCESS_TEMP}	
{ACCESSEE}	
NAME	EMP_NO
jim susan	1024 3189
jim susan	1024 3189
jim jim susan	1024 9999 3189
jim peter susan	9999 2705 3189

Fig. 3.2: ACCESS_TEMP ordered according to Definition 1

4. Alternatives to Sorting

Figure 3.2 indicates that sorting might provide a handle on the duplicate detection and deletion problem for the extended NF^2 data model. In this section we investigate whether there are alternatives to sorting.

4.1 Index Solution

An *index* is a mapping from attribute values to data objects. Consider a single attribute index for atomic values first. In order to decide whether a table is free of duplicates we could use the fact that if the index mapping is a function, then the indexed table is free of duplicates. Obviously, the inverse (an indexed table is free of duplicates only if it has an index with no multiple references) is false as can be seen from the following trivial example in Figure 4.1.

Neither `FILE_ID` nor `NAME` yields a key index on its own, yet the table has no duplicates. Still, existing index tables could be used to intersect the reference chains of length greater one between two indexes thus filtering out the only possible candidates. If no index tables exist, we could create them, say starting with the leftmost non-indexed attribute and proceeding to the next attribute as the need arises.

In the flat table `ACCESS_RIGHTS` from Figure 2.3, we could start with an index on `NAME`. For the entry `susan` we would have four references to tuples, for `peter` one, and for `jim` five references. Thus the tuple with `NAME = peter` could be eliminated as a candidate for duplicates. The remaining tuples would be selected into a temporary table and we could continue with an index on `EMP_NO`, either one for the whole temporary table or one per `NAME` value. This would split the set of tuples with `NAME = jim` into two subsets. Still, all reference chains per `EMP_NO` entry have length greater one and we had to look at the `FILE_ID` values for the tuples 'pointed to' in each chain. This would finally determine that the table is free of duplicates. Of course, a multi attribute index could also be used for that purpose.

The catch is that any intelligent sort would proceed in exactly the same way, i.e. it would never need to compare the `EMP_NO` fields of a tuple with `NAME = jim` with the `EMP_NO` field of a tuple with `NAME = susan`. Secondly, what if the table is `ACCESS_TEMP` in Figure 2.6? Now we need a complex object index. This is basically no problem, the index could conceptually be just a special NF^2 table with a 'pointer' attribute, but we are led back to the problem of detecting duplicates within the index!

Closely related to the idea of using an index with key property (i.e. where each entry 'points' to exactly one tuple) as a sufficient but not necessary condition for uniqueness, is the use of an additional attribute which will prove with high probability that the table is free of duplicates. As an example, consider an additional attribute `NO_OF_ACCESSEES` in Figure 2.5. It would record how many employees have access to each file, i.e. the cardinality of the set values. Given a projection on `ACCESSEE` (Figure 2.6), it would tell us immediately that only the first with the third tuple and the second with the fourth can be candidates for duplicates.

{ACCESS_RIGHTS}	
FILE_ID	NAME
customers	susan
customers	jim
accum-sales	susan
accum-sales	jim

Fig. 4.1: Attributes without key property but no duplicates

However, because it is only a sufficient indication of uniqueness, it does not solve our problem either. Moreover, cardinality of sets would be a data value stored internally by any NF² DBMS anyway and Definition 1 makes already use of this criterion. Unless a stronger 'signature' of tuples is used (cf. [FC84] for the use of signatures in document retrieval and the discussion of Hashing below), the use of artificial attributes under system or user control is a possible shortcut but not a solution in its own.

4.2 Hash-Function Solution

The idea is to specify a function f from the set V of (complex) values to, say, a set $A \subset \mathbb{N}$. If $A = \{1, \dots, m\}$, $f(v) = i$ can be used as index into a hash table of size m and f would act as a *hash function*. Only values v_1, v_2 which cause a collision, i.e. which map to the same address, would be candidates for duplicate detection. If the hash function were perfect (i.e. an injective function) [Sp77] all colliding values were guaranteed to be duplicates. If the hash function were both perfect and minimal [Ci80, Ja81, Chang84], m could be set to the cardinality of the NF² table. If the function f were not minimal and would even produce a very large (sparse) table A , we could still use f to map the (complex) values into a list A' of signatures and then sort A' to detect duplicates.

An essential requirement is that f be a function, i.e. two values which are identical in the NF² sense must map to one value in A . For sets the ordering of elements is immaterial, thus we need a function $f(a_1, \dots, a_n)$ which is invariant on the ordering of arguments a_1, \dots, a_n . Any commutative operator will do, and $f(a_1, \dots, a_n) = (a_1 + \dots + a_n) \bmod c$ for some constant c would be a candidate. If the elements a_i of the sets are not integers, an equivalent commutative operator must be used. For complex objects, the recursively computed hash values may be used as suggested by [KF88].

For lists and tuples order is relevant, i.e. $f(\langle 1, 2 \rangle)$ should not be identical to $f(\langle 2, 1 \rangle)$, resp. $f([1,2]) \neq f([2,1])$. Knuth [Kn73, p. 512] mentions multiword and variable length keys and notes that addition resp. exclusive *or* for folding the words makes use of all words in contrast to e.g. the mid-square technique for long character strings. Cichelli [Ci80] suggests using $h(k) = \text{length}(k) + g(\text{first character of } k) + g(\text{last character of } k)$. For the example above, i.e. the pair $f(\langle 1, 2 \rangle), f(\langle 2, 1 \rangle)$, this fails. In general, a non-commutative operator between the elements of the complex key should give better results. Candidates would be subtraction, division, exponentiation, remainder with a subsequent operation that leads back to the natural numbers. In [KF88], addition combined with shifting for each added key is proposed. The authors also suggest keeping the computed hash values as tags with the objects and examine marking and interfering as methods for avoiding repeated tests on duplicates.

However, hashing is not without pitfalls, as can be seen from the following example. Let a, b and c be pair-wise distinct complex objects and consider the set of sets $\{\{a, b\}, \{a, c\}\}$. Assume the unlikely, yet possible, coincidence where the hash function f yields $f(a) = f(b) = f(c) = k$. Thus the duplication detection algorithm must look inside a and b to determine that $\{a, b\}$ is indeed free of duplicates. The same happens for $\{a, c\}$. Both sets are marked "free of duplicates". On the next higher level, f uses the tagged hash values k yielding $f(\{a,b\}) = f(\{a, c\}) = k'$. Going again down one level, neither the tags nor the marking "free of duplicates" yield any clue as to whether $\{a, b\}$ is distinct from $\{a, c\}$ and we are back at the original problem.

Given a fixed table we could, in general, search for a perfect function, even a minimal one. This follows from Jaeschke's result [Ja81] which states that there always exist constants C, D, E s.t. $h(w) = \text{int}(C/(Dw + E) \bmod n)$, $n = |W|$ is a minimal perfect hashing function for a given arbitrary set W of positive integers. But it requires e.g. that the set W be sorted (!), is infeasible in the presence of many insertions and deletions or for large tables [Ja81] and is in any case computationally more expensive than sorting (Jaeschke reports 1.82^n iterations for n values to compute the minimal constant C as an experimental result).

Indeed, it is not necessary to use a minimal perfect hashing function to delete duplicates in minimal space without sorting. In [TW89a], a method is shown for detecting duplicates within an array and for moving these duplicates in place to the tail end of the array. The method requires in most instances several passes, but is independent of the hash function and has, on the average, a linear running time, i.e. it is much faster than sorting.

Hashing could also be used to test equality between sets. In [BKM86], Knuth suggests that equality of sets of integers be determined by inserting them into an ordered hash table. The two ordered hash tables are equal if and only if the two sets are equal. Since we must, in general, test equality between $n > 2$ sets, this would imply sorting the set of ordered hash tables. This seems less attractive than first sorting each set and then sorting the set of sets in some, yet unspecified, way. In summary, we see hashing as a very interesting speed-up which, however, must be backed up by a sorting method for degenerate cases. Whether it is ultimately easier to maintain a sorting order or a set of tagged hash values within typical NF^2 applications, further research and practical experience must show.

The final question is whether there is some method other than sorting, say some clever pairwise testing for equality, which would beat sorting. The answer is negative. For set equality, i.e. the question whether $A = \{A_1, \dots, A_n\}$ and $B = \{B_1, \dots, B_n\}$ are identical, Knuth ([Kn73], Sec. 5.3.3, Ex.23, p.209) mentions that under a suitable oracle any algorithm which only asks "Is $A_i = B_j$?" for certain i and j , must make at least $1/2 * n(n+1)$ comparisons in the worst case.

For duplicates in a set (or list), Aho, Hopcroft and Ullman [AHU83, p. 292] mention that to "purge duplicates from a list requires at least $\Omega(n \log n)$ time under the decision tree model of computation." A similar result for detecting the maximal repeated element in a list can be found in [MS76, p.5].

5. Considerations for a Sorting Solution

The previous section indicated that sorting seems to be the best choice for duplicate detection and deletion in the extended NF^2 data model. We know (cf. Section 2) that the process of deletion has to go from the inside to the outside. To describe which requirements an efficient algorithm should fulfill, we partially sort ACCESS_TEMP from Figure 2.6 according to the order relation from Definition 1. The output should be a permutation of the input with duplicates removed, i.e. a set ACCESS_TEMP whose elements are set valued tuples¹ ACCESSEE, where the set elements are tuples of degree 2.

5.1 Basic Principle and Example

We use a simple Selectionsort to keep our example free of unnecessary algorithmic detail. Thus, we proceed from the top of the table to the bottom, select the minimum and move it to the front (top). According to Def. 1, for tuples A and B , $A < B$ iff $FIRST(A) < FIRST(B)$, where $FIRST(X)$ for tuples X is defined s.t. the first attribute is most significant and the last (rightmost) attribute is least significant and where $MIN(X)$ for (multi)sets considers cardinality first. Note that we do not order (sort) objects inside more than needed. Sorting on all levels is in ascending order. Let $card(X)$ denote the cardinality of (multi)set X and define $fod(X)$ as a predicate which yields true iff object X is free of duplicates throughout. Note that we use the short-hand notation $\#i$ to identify the i 'th tuple in Figure 2.6 (left subtable) and that $card(\#i)$ denotes the cardinality of the set valued field of tuple $\#i$.

1. Pass (1. Minimum):

$\#1$ is considered min. Is there a smaller object among $\#2 - \#4$?

1 ACCESS_TEMP might actually be defined as a set of set of tuples, but a set of set valued tuples might be more intuitive for those readers who expect a relation to be a set of tuples.

1. Test (Level 1): #1 < #2? According to Def. 1 we should consider cardinality first. However, we can use card(#1), resp. card(#2), iff we know that fod(#1), resp. fod(#2), holds. Thus we must delete duplicates in #1 and #2, i.e. 'sort' recursively #1 and #2. Of course, we only do this once and then store the fact that the sets are fod.

Applying again Selectionsort, [susan, 3189] in #1 is considered minimum.

1. Test (Level 2): [susan, 3189] < [jim, 9999]? The answer is no because 'jim' < 'susan'. Thus [jim, 9999] is considered minimum.
2. Test (Level 2): [jim, 9999] < [jim, 1024]? The answer is no because FIRST([jim, 9999]\jim) = 9999 > 1024 = FIRST([jim, 1024]\jim). Thus [jim, 1024] is the smallest element in #1 and is moved to the front. For the next sub-pass within #1, [susan, 3189] is again considered minimum.
3. Test (Level 2): [susan, 3189] < [jim, 9999]? The answer is no and [jim, 9999] is moved to the front as 2nd minimum. This completes the ordering of #1 and because all tests between tuples either yielded 'strictly less' or 'strictly greater', we did not find duplicates and card(#1) = 3.

Similarly, #2 is 'sorted' with one test and yields card(#2) = 2. Now the test on the outer level can be answered, i.e. #1 > #2 because card(#1) > card(#2). Thus #2 is the new minimum.

2. Test (Level 1): #2 < #3? We know card(#2) = 2, but #3 is not proven fod. Thus #3 is 'sorted' first. In three tests we order #3, find no duplicates, and card(#3) = 3. Note that at this time only the EMP_NO values for [jim, 9999] and [jim, 1024] in #1 entered a test. All other EMP_NO values were never looked at! This changes with the next test when we discover the first duplicate.
 3. Test (Level 1): #2 < #4? Again, to determine card(#4), we sort #4. As a result we obtain card(#4) = card(#2) = 2. This implies that we must now look at successive minima in #2 and #4. The important observation here is that we should have stored the progress of the sort within each subset from previous passes. This way we can pick the minima without searching for them. The difference to storing the fod property is subtle and not well seen in this example. In other situations, however, a multiset might be marked fod but might not be completely ordered.
1. Test (Level 2): MIN(#2) = [jim, 1024] < [jim, 1024] = MIN(#4)? Since we note equality, we compare the second minima in both sets.
 2. Test (Level 2): MIN(#2\[jim, 1024]) = [susan, 3189] < MIN(#4\[jim, 1024]) = [susan, 3189]?

{ACCESS_TEMP}		
{ACCESSEE}		
NAME	EMP_NO	
#2	jim susan	1024 3189
	jim jim susan	1024 9999 3189
#3	jim peter susan	9999 2705 3189
#4 (duplicate)	jim susan	1024 3189

Fig. 5.1: ACCESS_RIGHTS ordered with duplicate deleted

Again the test yields equality, both #2 and #4 become empty and #4 (or #2) is identified and deleted as a duplicate. Subset #2 comes out as the minimum in the first pass and is moved to the front. Duplicate # 4 we move to the back.

The preceding Figure 5.1 shows the progress so far. Values in italics have entered a test at least once.

2. Pass (2. Minimum):

#1 is considered minimum. Is there a smaller object in the remaining 'list'?

1. Test (Level 1): #1 < #3? The answer is yes because $\text{card}(\#1) = 3 = \text{card}(\#3)$, but $\text{MIN}(\#1) = [\text{jim}, 1024] < [\text{jim}, 9999] = \text{MIN}(\#3)$. Thus #1 stays in front and duplicate elimination is completed. The final result is as in Figure 5.1 except that *[jim, 9999]* in #3 should now be *[jim, 9999]* because the EMP_NO value 9999 entered the test in this pass.

5.2 Discussion of the Example

The example above explained the task of deleting duplicates by sorting. On the other hand, it might not make the reader enough aware of the differences between

- sorting,
- sorting throughout,
- deleting duplicates, and
- deleting duplicates throughout.

In Figure 5.2 we give another, more contrived example. Applying the same algorithm as above (to find out whether two files are identical w.r.t. their sets of accessees), Figure 5.2 could act as input and Figure 5.3 (with values which entered the sort in italics) would be the output. We note that the result is sorted but not sorted throughout. It contains no duplicates on the top level but it is not free of duplicates throughout. The reason is that we never have to compare the attribute values for ACCESSEE, thus never compute their cardinalities and thus never delete their duplicates.

Whether the user expects duplicate deletion throughout in Figure 5.3 is a question of the application and the language interface. From the algorithmic point of view, a sorting solution should provide both alternatives: do a complete depth-first recursive sort or - usually less expensive - sort subobjects only to the level needed. The second alternative requires that the sort within subobjects can be halted, that the progress can be recorded without much extra space, and that the sort within subobjects can be continued from this point later on. A sort with this property is said 'to freeze well'.

Similarly, if objects are known to be free of duplicates but are not ordered (and need not be ordered more than what is required for duplicate detection on a higher level), then it would be an advantage to

{ACCESS_RIGHTS}	
FILE_ID	{ACCESSEE}
	NAME
trends	susan jim jim
budget	jim susan peter

Fig. 5.2: Input to sort

{ACCESS_RIGHTS}	
FILE_ID	{ACCESSEE}
	NAME
<i>budget</i>	jim susan peter
<i>trends</i>	susan jim jim

Fig. 5.3: Output of sort

have the minimum of a set with N elements delivered in $O(N)$ time. Since we may expect that objects differ already on the first MIN value, considerable savings would result for the sort on the outer level.

Another pleasant property would be elimination of duplicates 'on the fly'. As we enter a subobject to sort it, we might discover duplicates. If they can be eliminated right away, we save redundant comparisons and data fetches in later visits. Some sorting algorithms can be modified this way. In general, a sort tailored to multisets can achieve a speed up from $O(N \log N)$ to $O(N \log k)$ on inputs of size N with k distinct key values [MS76, We85].

The last criterion discussed here is 'stability'. According to Knuth [Kn73, p. 4], a sort is *stable* if equal keys retain their relative input order. We carry this term over to duplicate detection and deletion where it implies that we pick the first instance of several objects with equal keys from a given input sequence and that the list of duplicates retains the relative input order on equal keys. The second property allows repeated applications of the algorithm to detect 2nd, 3rd, ... instances of 'identical' objects in sequence.

Clearly, a stable sort can be used for stable duplicate detection and deletion. Conversely, we conjecture that any stable duplicate detection and deletion which uses sorting must employ a stable sorting algorithm. However, this seems hard to prove because one has to argue over the set of all conceivable sorting methods. Note that most unstable sorting methods, like Quicksort, can be made stable if extra space is provided. Even some hashing methods can yield stable duplicate deletion if extra space of order N is provided.

On the other hand, stable methods, like Mergesort and Selectionsort, can lose their stability if duplicates are swapped with non-duplicates at random. We therefore include a column 'in situ' in the following table to indicate whether an algorithm uses less or equal than $O((\log N)^2)$ bits of extra space. Table I reviews the better known sorting algorithms with respect to the criteria discussed so far. In the following, we point out the relative merits of each method and the necessary modifications.

5.3 Comparison of Suitable Sort Algorithms

Delivering the minimum in $O(N)$ steps and then the next $N-1$ minima in $O(\log N)$ steps each gives a total sorting time of $O(N \log N)$. The first three methods in Table I are well known quadratic sorts. However, for small 'lists' of very large complex objects, Selectionsort might be the method of choice because it requires only N data movements [Se83, p. 95]. The algorithm is easily modified to have a stable list of duplicates in the back and requires no extra space.

The Mergesort - workhorse of most database systems - suffers from the fact that the minimum is not available until the last runs are merged. Freezing the sort then is not worth the effort anymore: it only takes $O(N)$ steps more to complete the sort. Deleting duplicates on the fly and stable deletion are easy to incorporate and give a speed up [BD83]. The prize to be paid here is extra space, usually in the order of $N/2$.

Radixsort and Shellsort are not attractive at all. Quicksort can be modified to move duplicates to the back [TW89b]. However, it is not stable unless linked list versions are used. Also, it has large time gaps — up to N steps per gap — in delivering the next minimum even if the total sorting time is made $O(N \log N)$ in the worst case, say using a median-find algorithm. Moreover, it requires saving the recursion stack to freeze the sort; using a stackless variant [We87] would lead to a rather complicated and slow solution. All in all, if stability and/or some extra space are no issues, efficient external versions of Quicksort can be modelled after [SW84, TW89b] but we would not consider it first choice.

TABLE I

Method	MIN in $O(N)$	next in $O(\log N)$	freezes well	on-the- fly del.	stable	in situ
Selectionsort	yes	no	yes	yes	yes	yes
Insertionsort	no	no	yes	yes	yes	yes
Bubblesort	no	no	yes	yes	yes	yes
Mergesort	no	n.a.	no ¹	yes	yes	(no) ⁹
Radixsort	no	n.a.	yes	no	(no) ⁸	yes
Shellsort	no	n.a.	(no) ²	no ³	no	yes
Quicksort	yes ⁴	yes ⁴	(yes) ⁵	(yes) ⁶	(no) ⁸	yes
Heapsort	yes	yes	yes	yes ⁷	no	yes

Legend:

n.a. = not applicable

(1) difficult to freeze because of extra space, no sublist until after $O(N \log N)$ time

(2) easy to stop but sorted sublist hard to detect

(3) would have to handle variable increments

(4) only on the average

(5) requires storing the recursion stack unless stackless variants are used [We87]

(6) best in linked list [We85]

(7) efficiency yet unknown [TW89b]

(8) stable for list versions

(9) stable in situ methods are known but are considered impractical

Much to our surprise, Heapsort fulfills all criteria but one. Following the heap building phase, the minimum (maximum) is at the root. Using Floyd's bottom up building method [Fl64], this only takes linear time in the worst case. Thereafter, the next key in sort order is returned after each heapify (sink in) step which takes $O(\log N)$ comparisons in the worst case. To freeze the heapsort, we need only one pointer indicating the rightmost leaf, everything else is contained in the regular structure of the heap.

On-the-fly deletion requires some non-trivial modifications [TW89b]. Whenever the rightmost leaf is swapped with the root and sinks down the heap, we also test for equality. If an equal key is detected, the key is swapped on the spot with the rightmost leaf which may either continue to sink down or climb up. This causes hardly any overhead. On the other hand, duplicates are usually detected very late in the process of sorting, namely when they become adjacent at the root. Empirical tests with this version of Heapsort indicate that a speed up from $O(N \log N)$ to $O((N-M) \log (N-M))$ is possible, where M is the replication factor, i.e. $N = M \cdot k$. This is substantially slower than the speed up for Quicksort and Mergesort ($O(N \log N)$ to $O(N \log k)$). Moreover, Heapsort can also be turned into an efficient external sort [TW88] by using a heap of pages and merge operations between pages rather than exchanges of nodes.

The only obvious drawback with Heapsort is the fact that it is not a stable sort. If we insist on picking the first of several 'identical' objects, then it seems that a hidden attribute is the best solution. This attribute would record the index of the object relative to the input order. When objects with equal keys are detected, the 'index attribute' is compared and the object with smaller index value is kept. If the list of duplicates must be stable, we would then insert the other object into the list of duplicates using again the hidden index. Since Heapsort detects most duplicates as the root of the heap is removed, the insertion path is short.

Another alternative is to store the list of removed unique objects separately. This gives rise to a 'heap with holes' as described in [TW88]. Using a different arrangement for the heap we are then able to modify the sort to become stable at the expense of extra space. The resulting method now resembles Tree selection sort [Kn73, p. 142].

In summary, *Mergesort* is a safe choice if tables are always sorted throughout and if extra space is no issue. If objects need not come into physical sequence, linked list versions could be used. The modifications required for stable deletion are easy to incorporate. *Quicksort* in a stable linked list version for sorting throughout would be hard to beat performancewise. Special precaution has to be taken for the worst case. The necessary modifications can be taken from the literature. If sorting throughout is not required, *Heapsort* is best suited for the given task within the extended NF^2 data model. If stability is required, an extra attribute must be added. The necessary modifications for in situ duplicate deletion are outlined above but are not trivial.

6. Conclusion

Detecting and deleting duplicates in the extended NF^2 data model is a complex task both in terms of what needs to be offered to the user and in terms of implementation. Having introduced the notion of 'uniqueness' for tables, sets and lists in the extended NF^2 model, we introduced three ordering relations for complex objects. The definition which uses cardinality of sets and repeated minima as ordering criterion was picked as basis for the following discussion. We then reviewed existing and newly created indexes as well as hashing as possible alternatives to sorting for the given task. However, while both approaches can be used as a short-cut in many situations, they offer no general solution. The main reason is that duplicate deletion must handle complex objects which may contain duplicates themselves.

The recursive nature of the task is then shown in a detailed example where we apply a simple Selectionsort to a table with a set-valued attribute. The example illustrates that objects need to be sorted only up to the point where inequality can be detected. This gives rise to the notion of 'freezing' a sort within subobjects. Furthermore, the availability of the minimum in linear time rather than at the end of the sort is identified as a useful property. Finally, stability of duplicate deletion is introduced and is related to stable sorting methods.

Given these requirements, the popular sorting methods are reviewed and a modified Heapsort is selected as method of choice. As shown, Heapsort fulfills all criteria but stability which must be handled by a hidden attribute. At present, a preliminary implementation of a recursive Heapsort for nested complex objects is being tested. Further experience must show whether this unusual choice is justified.

References

- AB84** Abiteboul, S., Bidoit, N.: Non First Normal Form Relations: An Algebra Allowing Data Restructuring. *Rapports de Recherche No. 347*, Institut de Recherche en Informatique et en Automatique, Rocquencourt, France, Nov. 1984.
- AHU83** Aho, A.V., Hopcroft; J.E., Ullman, J.D.: *Data Structures and Algorithms*, Addison-Wesley, Reading, Mass., 1983.
- ALPS88** Andersen, F., Linnemann, V., Pistor, P., Südkamp, N.: *Advanced Information Management Prototype: User Manual for the Online Interface of the Heidelberg Database Language (HDBL) Prototype Implementation, Release 2.0*, Technical Note TN 86.01, IBM Heidelberg Scientific Center, Jan. 1988.
- BD83** Bitton, D., DeWitt, D.J.: Duplicate record elimination in large data files, *ACM Trans. Database Syst.*, June 1983, pp. 255-265.
- BKM86** Bentley, J., Knuth, D.E., McIlroy, D.: Programming Pearls: A Literate Program, *Comm. ACM*, Vol. 29, No. 6, June 1986, pp. 471-483.
- Ch81** Chamberlin, D.D., et al.: Support of Repetitive Transactions and Ad Hoc Queries in System R., *ACM Transactions on Database Systems*, Vol. 6, No. 1, March 1981, pp. 70-94.

- Chang84** Chang, C.C.: A study of an ordered minimal perfect hashing scheme, *Comm. ACM*, Vol. 27, No. 4, April 1984, pp. 384-387
- Ci80** Cichelli, R.J.: Minimal Perfect Hash Functions Made Simple, *Comm. ACM*, Vol. 23, No. 1, Jan 1980, pp. 17-19.
- Co70** Codd, E.F.: A Relational Model of Data for Large Shared Data Banks, *Comm. ACM*, Vol. 13, No. 6, June 1970.
- Da81** Date, C.J.: *An Introduction to Database Systems* (3rd ed.), Addison-Wesley, Reading, Mass., 1981.
- Da86** Dadam, P., et al.: A DBMS Prototype to Support Extended NF² Relations: An Integrated View on Flat Tables and Hierarchies, *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Washington, D.C., May 1986, pp. 356-367.
- DGK** Dayal, U., Goodman, N., Katz, R.H.: An Extended Relational Algebra with Control Over Duplicate Elimination, *Proc. ACM Symp. PoDS*, Los Angeles, Cal., March 1982, pp. 117-123.
- FC84** Faloutsos, C., Christodoulakis, S.: Signature Files: An Access Method for Documents and its Analytical Performance Evaluation, *ACM TOOIS*, Vol. 2, No. 4, Oct. 1984, pp. 267-288.
- Fl64** Floyd, R.W.: Algorithm 245, Treesort 3, *Comm. ACM*, Vol. 7, No. 12, Dec. 1964, p. 701.
- Ja81** Jaeschke, G.: Reciprocal Hashing: A Method for Generating Minimal Perfect Hashing Functions, *Comm. ACM*, Vol. 24, No. 12, Dec. 1981, pp. 829-833.
- JS82** Jaeschke, G., Schek, H.-J.: Remarks on the Algebra of Non First Normal Form Relations, *Proc. ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Los Angeles, Cal., March 1982, pp. 124-138.
- KF88** Khoshafian, S., Frank, D.: Implementation Techniques for Object Oriented Databases; in: *Advances in Object-Oriented Database Systems*, K.R. Dittrich (Ed.), Springer LNCS 334, Sept. 1988, pp. 60-79.
- Kn73** Knuth, D.E.: *The Art of Computer Programming*, Vol. 3: Sorting and Searching, Addison-Wesley, Reading, Mass., 1973.
- MS76** Munro, I., Spira, P.M.: Sorting and Searching in Multisets, *SIAM J. Comput.*, Vol. 5, No.1, March 1976, pp. 1-8.
- PA86** Pistor, P., Andersen, F.: Designing a Generalized NF² Data Model with an SQL-Type Language Interface, *Proc. 12th Int. Conf. on Very Large Data Bases*, Kyoto, Japan, Aug. 1986, pp. 278-288
- Pi87** Pistor, P.: The Advanced Information Management Prototype: Architecture and Language Interface Overview, *Proc. Troisièmes Journées Bases de Données Avancées*, Port Camarque, France, May 1987 (invited paper).
- PT86** Pistor, P., Traummüller, R.: A Database Language for Sets, Lists and Tables, *Information Systems*, Vol. 11, No.4, 1986, pp. 323-336.
- Ro85** Roth, M.A.: SQL/NF: A Query Language for \rightarrow NF Relational Databases, Technical Report TR-85-19, Univ. of Texas at Austin, Dept. of Computer Science, Sept. 1985.
- Se83** Sedgewick, R.: *Algorithms*, Addison-Wesley, Reading, Mass., 1983.
- SJ77** *IBM Systems Journal*, Special Issue on IMS, Vol. 16, No. 2, 1977.
- SLPW89** Saake, G., Linnemann, V., Pistor, P., Wegner, L.: Sorting, Grouping, and Duplicate Elimination in the Advanced Information Management System, IBM Heidelberg Scientific Center (in preparation).
- Sp77** Sprugnoli, R.: Perfect hashing functions: A single probe retrieving method for static sets, *Comm. ACM*, Vol. 20, No. 11, Nov. 1977, pp. 841-850.
- SS86** Schek, H.-J., Scholl, M.: The Relational Model with Relation-Valued Attributes, *Information Systems*, Vol. 11, No. 2, 1986, pp. 137-147.
- St76** Stonebraker, M., et al.: The Design and Implementation of Ingres, *ACM Trans. on Database Systems*, Vol. 1, No. 3, Sept. 1976, pp. 189-222.
- SW84** Six, H.W., Wegner, L.: Sorting a Random Access File in Situ, *Computer Journal*, Vol. 27, No. 3, pp. 270-275, 1984.
- TW88** Teuhola, J., Wegner, L.: The External Heapsort, *IEEE Trans. Softw. Eng.*, 1988 (in print).
- TW89a** Teuhola, J., Wegner, L.: Linear Time, Minimal Space Duplicate Deletion, *Math. Schriften Kassel*, No. 2/89, January 1989.
- TW89b** Teuhola, J., Wegner, L.: A tale of sorts: duplicate deletion in Quicksort, Mergesort and Heapsort, 1989 (in prep.).
- We85** Wegner, L.: Quicksort for Equal Keys, *IEEE Trans. on Computers*, Vol. C-34, No. 4 (April 1985), pp. 362-367.
- We87** Wegner, L.: A Generalized, One-Way, Stackless Quicksort, *BIT*, Vol. 27, No. 1, pp. 44-48, 1987.