Multirelations — Semantics and Languages

Aviel Klausner
Harvard University

Nathan Goodman
Sequoia Systems

Abstract

We argue that a multirelation (relation with duplicates) is not a semantically independent data object, but rather it should be viewed as a subset of columns within a larger relation that has no duplicates. Consequently, at the level of the conceptual database, duplicates in base relations or in views are not allowed, nor are operations on multirelations. Multirelations as query output can be specified by designating a subset of some relation's columns for output, while "hiding" the rest of the columns. Similarly, aggregate functions are applied to multirelations by applying them to a column within a relation. Our approach can be applied to extend any query language in a consistent way to have full multirelational expressiveness, and such an extension for the query language QUEL is detailed.

1 Introduction

The problem of duplicate tuples is usually ignored by the traditional relational model theory. By definition relations are sets, so the same tuple cannot exist more than once in a relation. This approach has the advantage of being mathematically clean and elegant. However in practice the need for relations with duplicates, or *multirelations*, can arise. For example, a user may pose a query whose answer could have duplicates, such as a query requesting the names of a certain group of people. The list may contain duplicates if several people have the same name, and the number of duplicate names may be significant to the user.

DAPLEX [Ship81], a query language based on the entity-relationship model, is an example of a language that specifically supports multirelational queries, i.e. queries which return duplicates. Its iteration-based semantics gives queries flexible control over the creation and elimination of duplicates. In PROLOG [Hogg84], which has aspects of a relational query language, the predefined operational semantics sometimes forces the output to be repeated a specific number of times. However many other database systems, in coping with duplicates, are inconsistent and often ill-defined. For example, the number of duplicates in the output of a QUEL query, as implemented in INGRES [WY79], might depend on the access strategy chosen for evaluating the query. The user has no control over the elimination of duplicates in this output. The UNIQUE (') feature in QUEL gives limited control over duplicate elimination when applying aggregate functions, however it cannot be used to eliminate duplicates in the output of a query.

The first attempt to incorporate duplicates into the theory was presented in [DGK82], motivated by the need to process multirelational queries in DAPLEX. The relational model was generalized by replacing the relation with the more general multirelation (*multiset relation* in [DGK82]), which is a relation with duplicates. In the generalized *multirelational model* the database consists, in general, of multirelation instances, which are the main data objects. *Multirelational algebra* was defined as a generalization of relational algebra to enable the manipulation of multirelations by algebraic operators. The input as well as the output of a query

consist, in general, of multirelations, of which relations are a special case.

The multirelational model provides a mathematically elegant generalization of the relational model, and it is a practical formalism for handling multirelations. On this issue we disagree with Klug [Klug82], which ignores the need for multirelations as query output, a need demonstrated by our examples. Contrary to his claim, multirelational queries can be optimized, and the multirelational model is useful in processing such queries, as shown in [Klau85]. For example, a multirelational expression involving the union of two restrictions on the same relation, $(\sigma_P R) \cup (\sigma_Q R)$, was given in [Klug82]; it is true that the standard access strategy that scans R and outputs tuples that satisfy either P or Q will not preserve the number of duplicates in this expression as required. Nevertheless, this can be "fixed" in a simple way — a tuple that satisfies both conditions would be output twice.

Although the multirelational model is useful in the *internal* levels of the database for processing multirelational queries, it does not provide an adequate semantic framework for understanding the role of multirelations at the *conceptual* level, i.e. the abstract representation of the data in the database. This paper proposes such a framework, which addresses questions such as: When do duplicates occur, what do they represent, should base relations or views contain duplicates, and how can a query specify precisely the extent of duplicates in its output?

According to our approach, which is developed and justified in Section 3, at the conceptual database level a multirelation is not a semantically independent data object with complete data. In the entity-relationship model of the world (see [Ullm82]) a relation represents a set of entities in the real-world, one tuple per entity. A multirelation also refers to a set of distinct real-world entities, but it contains only partial information about these entities, thus giving rise to duplicates. The multirelation consists of a subset of columns within some relation without duplicates that represents these entities. This relation is called a *complete relation*, its columns that appear in the multirelation are called *output* columns, and those missing are called *hidden* columns. A multirelation can be considered a projection of a complete relation on the output columns without eliminating duplicates. As a consequence of this approach, base relations or views cannot contain duplicates, and manipulating multirelations as such should be avoided at the conceptual level.

A query that requests a multirelation as output actually inquires about the entities of the complete relation, while it needs only part of the information about these entities, namely the output columns. Hence such a query should be able to specify the complete-relation-entities, and to designate any subset of its columns as the output multirelation. Accordingly, Section 4 describes how any query language can be extended consistently to have full multirelational expressiveness, and in Section 5 such an extension for the query language QUEL is detailed.

Our approach to multirelations helps to understand another instance in which one should be aware of duplicates, namely in the input to an aggregate function. Addressing this problem Klug [Klug82] correctly observed that aggregate functions should be applied to a column (or set of columns) within a relation that has no duplicates. This is clearly justified by our semantic approach: A complete relation represents the desired set of entities, and the aggregate function is applied to the output column within this relation while the rest of the columns are hidden. Thus aggregate function input can be treated as a particular use of multirelational queries, and indeed in Section 5 it is shown how the exact same constructs used to extend QUEL with multirelational queries are useful in supporting generalized aggregate functions as well, in accordance with the suggestions in [Klug82].

2 Basic Definitions

By the standard definition a relation cannot contain duplicate tuples, as it is a set. Collections of tuples in which this requirement is relaxed are called multirelations, i.e. multisets of tuples.

Formally, a *universe* U is a set of *attributes*, each associated with a *domain*. A *relation scheme* $\underline{R}$ is a subset of attributes, and a *database scheme* is a collection of relation schemes. A *tuple* t over $\underline{R}$ is a mapping from $\underline{R}$'s attributes into their respective domains. A *set relation instance* (or simply a *relation*) over $\underline{R}$ is a set of tuples over $\underline{R}$, while a *multirelation instance* over $\underline{R}$ is a *multiset* of tuples over

$\underline{R}$, i.e. an unordered collection of (possibly multiple copies of) tuples. $R(A_1, \ldots, A_k)$ is often used to denote that R is a (multi)relation over $A_1 \ldots A_k$, which are also called the *columns* of R. The number of copies of t in a multirelation R is called the *multiplicity* of t in R and is denoted by $\#(t, R)$. A relation is the special case in which $\#(t, R) = 1$ for all $t \in R$. Note that, strictly speaking, a multirelation R over $\underline{R}$ is a mapping from the domain of $\underline{R}$ into Z.

In the *entity-relationship model of the world* (see [Ullm82]) the real world is modeled by sets of *entities* of various sorts together with some *relationships* among these entities. In the traditional *relational model of data* this information is represented by relations. Each relation can represent some set of real-world entities of a similar type, where each tuple represents one entity and each attribute represents some characteristic of this type of entities.[1] The data resides in relation instances, and the *database instance* is a collection of relation instances (also called *base relations*). Extracting information from a database is done using *queries*, which may be regarded as mappings from database instances to relation instances. *Relational algebra* is a language for expressing such queries, and it consists of a set of operations for manipulating relations.

In the generalized *multirelational model* [DGK82], the database consists, in general, of multirelation instances, which are the main data objects. The input as well as the output of a query consist, in general, of multirelations which might therefore contain duplicates. *Multirelational algebra* [DGK82,Klau85] is a generalization of relational algebra which enables the manipulation of multirelations by algebraic operators. For example, the standard *project* operation is separated into two distinct operations: *Project without eliminating duplicates* (Π), which might result in a multirelation, and *unify* (δ), which merges identical tuples into a single copy, thus transforming a multirelation into a relation.

[1] A relation can also represent a *relationship* between entities, but for our purpose the set of relationship instances itself can be considered a set of entities.

3 A Semantic Approach to Multirelations

The multirelational model gracefully generalizes the relational model, introducing duplicates into relational database theory. It provides a practical formalism for handling multirelations and for processing multirelational queries, yet at the conceptual level it does not provide an adequate semantic framework for understanding duplicates. This aspect of the role of multirelations in databases is examined in this section, and a semantic framework is proposed that treats multirelations within the relational model, without need to generalize the model. We use the following example throughout this paper to demonstrate these ideas.

Let **ASSIGNMENT(emp#, name, salary, dept)** be a relation describing employees and their assignments to departments, where each employee can work in one or more departments. Assume that a list of names of employees working in either the Toy or the Hat departments is requested.

A careful reader would probably notice that **ASSIGNMENT** is not normalized, and in the database it might be decomposed into the relations **E(emp#, name, salary)** and **A(emp#, dept)**. Nevertheless, the given query ould then be naturally solved by first joining **E** and **A** over **emp#** to create the relation **ASSIGNMENT**.

Clearly, the requested list of names would be taken from the **name** column of **ASSIGNMENT**, after selecting tuples which refer to either of the two departments. However, projecting $\sigma_{name \in (TOY,HAT)}$**ASSIGNMENT** on the **name** column and eliminating duplicates would yield an incorrect answer since two employees can have the same name, and each one of them should be output separately. Projecting on the **name** column without eliminating duplicates is also wrong, since the name of an employee working in both departments would appear twice.

The correct way is to create a new relation **EMP(emp#, name)**, by projecting $\sigma_{name \in (TOY,HAT)}$**ASSIGNMENT** on these two columns and eliminating duplicates. **EMP** has exactly one tuple for each employee that works in either department, as specified in the problem. The **name** column of **EMP** constitutes the requested

multirelation, which therefore can be produced by "projecting" **EMP**, without eliminating duplicates, on the **name** column.

Although this multirelation consists only of names, it is related to the employees in **EMP** in a very direct way: It has exactly one tuple for each employee. In other words, the multirelation actually pertains to the **EMP**-entities, and its duplicates arise precisely because some tuples in **EMP** refer to different employees with the same name.

Going back to the entity-relationship view of the world we can say that in general, for each multirelation M over X there exists a set of entities E to which M refers, such that there is a 1-1 correspondence between the tuples of M and the entities in E. M contains only partial information about each entity, namely the attributes X. These attributes might not be enough to uniquely identify each entity, thus yielding duplicates in M. The entities in E can however be described by some relation without duplicates R over some set of attributes XY, such that R contains enough information to uniquely identify each entity in E. R is called a *complete relation* for M, its columns that appear in M are called *output* columns, and those missing are called *hidden* columns. M constitutes a subset of R's columns, and indeed M is usually created by projecting a complete relation R on the output columns X without eliminating duplicates.

We contend that a query requesting a multirelation as output actually enquires about the entities of the complete relation, while only part of the information about these entities, namely the output columns, is needed. The query specifies the set of entities to be described by the complete relation, which in our example is the specified set of employees. In addition it specifies what is of interest about these entities, that is, which columns in this complete relation would be output and which would be hidden. In our example only the names of these employees are needed and the employee-numbers are hidden.

As a consequence, a multirelation, which only partially describes the entities it represents, cannot serve as an independent data object in the database, i.e. a base relation (or a view) should not contain duplicates. Semantically a base relation is supposed to describe some real world entities (or relationships). All the relevant information about them is supposed to reside in it, and in particular it should contain all the information required to identify and distinguish between the various entities. Indeed if several real world entities have exactly the same values for all the attributes in a base relation, then the relation scheme should be purposely augmented by attributes that distinguish between the entities in the real world.[2] On the other hand, if that distinction is irrelevant, then the relation is not really describing these entities, but rather it is describing some "collection of properties" these entities can have. In either case the relation has no duplicates.

Similarly, multirelations should not be manipulated as independent data objects, at least not at the level of the conceptual database. Manipulating multirelations can be useless or misleading, since in general multirelations contain only partial information about the complete relation entities, and the information lost might be needed later, as illustrated by the following example.

Let **TOY-NAME(name)** and **HAT-NAME(name)** be two unary multirelations, with complete relations **TOY-EMP(emp#, name)** and **HAT-EMP(emp#, name)**, respectively, containing information about the employees in each respective department. It would seem natural to derive the list of names requested above by combining the two multirelations. However, this task is impossible since there is no way to know whether a name appearing in both multirelations refers to the same employee or not. In fact the requested multirelation can range anywhere between **TOY-NAME** *max* **HAT-NAME** and **TOY-NAME** + **HAT-NAME**[3] (of course, eliminating all duplicates is wrong since two employees in the same department might have the same name). The correct answer is given by creating the union of the complete relations **TOY-EMP** ∪ **HAT-EMP** (without duplicates), thus a person who works in both departments is considered in this relation only once. The **name** column within this relation is the requested multirelation.

It is important to emphasize that designating some columns of the complete relation as output and hiding the rest is conceptually different from

[2] A common way of doing this is by adding an arbitrary ID column to the relation scheme.

[3] These multirelational algebraic operations are defined formally in [Klau85].

the relational algebraic project operation. When a projection of a relation is taken, a new and independent relation is created. This relation might describe different entities than those described by the original relation. In contrast, the multirelation which consists of the output columns refers exclusively to the entities of the complete relation, and therefore it has no meaning by itself. For example, while the multirelation Π_{salary}**EMP** refers to the entities of EMP and contains salaries of individual employees, the relation $\delta\Pi_{salary}$**EMP**, with no duplicates, does not refer to employees but to salary entities, i.e. it describes the various salary figures paid by the employer.

Our semantic approach to multirelations helps to understand another instance in which one should be aware of duplicates, namely in the input to an aggregate function. The value of many aggregate functions, e.g. *Count*, *Sum*, *Average*, depends directly on the existence or elimination of duplicates in the argument. Addressing this problem Klug [Klug82] correctly observed that aggregate functions should be applied to a column (or set of columns) within a relation that has no duplicates to produce a single value. This is clearly justified by our approach: An aggregate function is applied to some set of real-world entities using part of the information about each entity, usually a single attribute. In our terminology we can say that a complete relation represents this set of entities, a subset of output columns within this relation serves as the aggregate function argument, and the rest of the columns are hidden. Thus aggregate function input can be treated as a particular use of multirelational queries.

To summarize, duplicates should be incorporated into databases not by viewing multirelations as a generalization of relations, but rather by explaining them within the relational framework as a subset of some relation's columns. A multirelation can serve as query output or as an aggregate function argument only as part of a complete relation, but it cannot be manipulated as an independent data object.

4 Application to Languages

Using the above concepts we can now describe specifically what is required of a relational database language to be able to express multirelational queries. Since we concluded that base relations cannot contain duplicates, there is no need to change the data definition language (DDL). As for queries in the data manipulation language (DML), it suffices to specify the construction of the complete relation without duplicates, followed by the hiding of some of its columns, thereby creating duplicates in the output. A query language that can eliminate duplicates in any constructed relation and can output any subset of its columns while hiding the rest is said to have *full multirelational expressiveness*.

For any given query language the construction of the complete relation can be specified without any change to the language. To this should be added the ability to specify which of its columns would be in the output multirelation and which would be hidden; the trivial case of no hidden columns corresponds to standard queries with duplicate elimination. In specifying the complete relation each language preserves its individual methodology and expressive capabilities, and adding the ability to create from this relation some output with duplicates gives the language full multirelational expressive power in a natural and consistent way.

Domain Relational Calculus [Ullm82], for example, can be extended accordingly in a trivial manner, by dividing the free variables into two groups, specifying which correspond to the output columns and which to the hidden columns:

$$< x_1, \ldots, x_n >: \; < h_1, \ldots, h_p >: \\ F(x_1, \ldots, x_n, h_1, \ldots, h_p).$$

In the example above the multirelation of names of employees working in either the Toy or the Hat departments would then be given by the expression

$$n : e : \exists s(\; \mathbf{ASSIGNMENT}(e, n, s, TOY) \vee \\ \mathbf{ASSIGNMENT}(e, n, s, HAT) \;).$$

5 Extended QUEL

QUEL [HSW75] is a tuple calculus query language developed for the relational database system INGRES. The behavior of duplicates in QUEL is mostly undefined. In the implementation described in [WY79] QUEL queries are evaluated using tuple substitution, which is similar to the

depth-first execution and unification [Hogg84] used in PROLOG. This method has the same effect as nested loops, one loop per relation mentioned, causing duplicates in the output of both languages, since each tuple is constructed and accepted for output independently of the other tuples. In both languages duplicates are not always desirable, but while in PROLOG the creation of duplicates is precisely specified by the operational semantics of the language, no such definition exists in QUEL. Indeed, the number of duplicates in QUEL is inconsistent and might depend on the access strategy chosen to evaluate the query. For example, the elimination of a redundant clause during optimization might reduce the number of duplicates in the output. One exception is the **UNIQUE** feature, which provides limited control over duplicate elimination when applying aggregate functions, yet it cannot be used to eliminate duplicates in the output of a query. Thus the user has no control over the elimination of duplicates in this output.

In the following we extend QUEL to have precise and full multirelational expressiveness. The extension also introduces into QUEL generalized aggregate functions, in accordance with the suggestions in [Klug82], as explained above.

Formally, a QUEL retrieval query has the following form (square brackets designate an optional part):

RETRIEVE [**INTO** R] $(t_1,\ldots,t_n)$ **WHERE** q

The qualification q contains tuple variables that correspond to base relations, and the target-list $t_1,\ldots,t_n$ contains attribute terms obtained from these variables. The output contains a tuple $< t_1,\ldots,t_n >$ iff it corresponds to some q-satisfying tuple in the cartesian product of all the relations.

Using the example above, the list of employees in either the Toy or the Hat departments is requested. The query

```
RANGE OF A IS ASSIGNMENT
RETRIEVE (A.name) WHERE (A.dept = 'Toy')
OR (A.dept = 'Hat')
```

does not work, since an employee of both departments has two tuples in **ASSIGNMENT** that satisfy the qualification, and therefore the name would be listed twice.

To enable multirelational queries according to the concepts developed above, the query is regarded as specifying the complete relation, and the target list is divided into two parts, specifying the output columns and the hidden columns within the complete relation:

RETRIEVE $(t_1,\ldots,t_n$ [**HIDING** $h_1,\ldots,h_p$]) **WHERE** q

where the h_i are attribute terms. The qualification q together with all the terms in the target list specify the complete relation, and each t_i (resp. h_i) specifies an output (resp. hidden) column. The tuple $< t_1,\ldots,t_n >$ appears in the multirelation output once for each tuple $< t_1,\ldots,t_n,h_1,\ldots,h_p >$ that satisfies the qualification. Omitting the **HIDING** part will result in the elimination of all the duplicates in the output. In Extended QUEL the query requested in the example is:

```
RETRIEVE (A.name HIDING A.emp\#) WHERE
(A.dept = 'Toy') OR (A.dept = 'Hat')
```

Notice that the **HIDING** option should be allowed only in cases in which the **INTO** clause is not used. As discussed above, it is not appropriate to create multirelations as named objects to be manipulated later. An alternative could be to allow the creation of these named multirelational objects, but to distinguish them from relations and to accordingly restrict their use and manipulation.

This extension for multirelational queries can also be applied to improve the aggregate capabilities of the language in a consistent way. The basic form of aggregates in QUEL is:

$$AGG(t \textbf{ WHERE } q)$$

The aggregate function AGG is applied to a one-column relation which is the output of a query with target-list t and qualification q. That "relation" is in fact a multirelation, since by default duplicates are not eliminated before applying AGG.

Assume we want to compute the total salaries of the employees in **ASSIGNMENT**. The expression

```
SUM(A.salary)
```

gives an incorrect answer, since it sums twice the salary of a person working in two departments.

The **UNIQUE** feature (sometimes denoted ') was added to QUEL to enable the elimination of duplicates before applying aggregate functions such as **COUNT, SUM, AVG.** The aggregation then has the form:

$$AGG \textbf{ UNIQUE } (t \textbf{ WHERE } q)$$

Unfortunately even with the **UNIQUE** feature QUEL does not provide full control over duplicate elimination in aggregate function input. When **UNIQUE** is used, all the duplicates are eliminated before applying the aggregate function. As we have noted above, in some cases neither full elimination nor no elimination of duplicates are correct. In our example the expression

```
SUM UNIQUE(A.salary)
```

sums only once a salary figure earned by two employees. (The only way the correct answer can actually be obtained in QUEL is by creating a new relation **E(emp#, salary)**, changing its physical structure so that duplicates will be eliminated, and then summing its salaries.)

As mentioned above, the problem can be solved by having aggregate functions range over all the entities of a complete relation, while applying them only to one of its columns, as suggested in [Klug82]. In QUEL this can be achieved using exactly the same construct we have used to implement multirelational queries. The target-list t is modified, adding to it the specification of the hidden columns, in exactly the same manner as was done for the target-list of a general query. The new aggregation has the following more general form:

$$AGG(t \textbf{ [HIDING } h_1, \ldots, h_p\textbf{] WHERE } q)$$

The hidden terms $h_1, \ldots, h_p$, if specified, together with the term t form a complete relation. The duplicates in this relation are eliminated, and then the multirelation that consists of column t is taken as the argument for AGG. If no duplicate elimination is desired, keys to all the relations mentioned in the query should be listed as hidden terms. All the duplicates are eliminated if no hidden terms are specified, thus obviating the **UNIQUE** feature. Any degree of duplicate elimination in between is possible, for example the value requested above is produced correctly by the following expression:

```
SUM(A.salary HIDING A.emp\#)
```

Our extension for aggregation is also useful when a **BY** clause is used to partition a set of tuples and to apply an aggregate function to each partition separately. For example,

```
COUNT(A.emp\# BY A.dept)
```

counts the number of employees in each department separately. This feature is also suggested in [Klug82] as part of a generalized aggregate function capability.

To enable the full control over duplicate elimination when using the **BY** clause the same construct as above can be used:

$$AGG(t \textbf{ BY } t_1, \ldots, t_n \textbf{ [HIDING } h_1, \ldots, h_p\textbf{]} \textbf{ WHERE } q)$$

All the attribute terms $t, t_1, \ldots, t_n, h_1, \ldots, h_p$ form a complete relation, and its duplicates are eliminated. Then the tuples are partitioned according to their values on the grouping terms $t_1, \ldots, t_n$, and AGG is applied to the t column of each partition separately. Each set of values of the grouping terms is output together with the respective result of AGG. For example, the average salary in each department is computed by the following expression:

```
AVG(A.salary BY A.dept HIDING A.emp\#)
```

6 Conclusions and Related Work

In this paper we have discussed the semantic aspects of duplicates, providing a conceptual framework for understanding their role in query output as well as in aggregate function input. It was argued that, semantically, multirelations should be viewed as some subset of columns within a relation without duplicates. A multirelation is not a semantically independent data object, since it contains only partial information about the real world entities to which it refers, and therefore it is not appropriate for base relations or views to contain duplicates. Multirelations are important as query output, when only partial information about some set of entities is requested for output. They are similarly important for aggregate functions, which

are usually applied only to a single attribute of the set of entities.

We have shown how these concepts can be applied to extend any relational query language in a simple and natural way to have full multirelational expressiveness. A QUEL extension was presented that supports full control over duplicates in query output. This extension also supports in a uniform manner control over duplicates in aggregate function input in accordance with the ideas presented in [Klug82].

To aid in the optimization of multirelational queries in the query processing stage a tableau formalism was defined in [DGK82], generalizing the method of [ASU79]. Tableaux are useful for checking equivalence among such queries and for simplifying them, as shown in [Klau85].[4] The Chase process [MMS79], for simplifying tableaux when functional dependencies are present, is also extended for multirelational queries in [Klau85].

Although not appropriate at the level of the conceptual database, the multirelational model and multirelational algebra do prove useful in the actual processing of multirelational queries, since the manipulation of multirelations instead of relations can then be advantageous. For example, the multirelational algebraic expression $\Pi_A(AB \bowtie BC \bowtie CD)$ is equivalent to the expression $\Pi_A(AB \bowtie \Pi_B(BC \bowtie \Pi_C CD))$. The second expression might be easier to evaluate using semijoin operations instead of joins, with counters in the physical representation of a multirelation instead of the duplicates themselves. It would be useful to extend the theory of database scheme acyclicity in order to identify the cases in which such a technique is helpful.

Acknowledgements

We would like to thank Larry Denenberg and Ed Sciore for commenting on earlier versions of this paper. The discussions with Christoph Freytag and his continuous encouragement were indispensable during the making of this paper. The work was supported in part by National Science Foundation grants mcs–79–07762 and mcs–82–01429.

[4] We should point out that as stated, Theorems 3.2 and 3.3 in [DGK82] are incorrect. However, it is possible to modify the definition of containment mapping there to enable tableau equivalence checking and tableau simplification.

References

[ASU79] Aho, A. V., Y. Sagiv, and J. D. Ullman, "Equivalences Among Relational Expressions," SIAM Journal of Computing, Vol. 8, No. 2, May 1979, pp. 218–246.

[DGK82] Dayal, U., N. Goodman, and R. H. Katz, "An Extended Relational Algebra with Control Over Duplicate Elimination," Proc. ACM Symp. Principles of Database Systems, 1982, pp. 117–123.

[HSW75] Held, G. D., M. R. Stonebraker, and E. Wong, "INGRES — A Relational Database Management System," Proc. AFIPS NCC, Vol. 44, May 1975, pp. 409–416.

[Hogg84] Hogger, C. J., "Introduction to Logic Programming," Academic Press, 1984.

[Klau85] Klausner, A., "Multirelations in Relational Databases," Manuscript, 1985.

[Klug82] Klug, A., "Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions," Journal of the ACM, Vol. 29, No. 3, July 1982, pp. 699–717.

[MMS79] Maier, D., A. O. Mendelzon, and Y. Sagiv, "Testing Implications of Data Dependencies," ACM Transactions on Database Systems, Vol. 4, No. 4, December 1979, pp. 445–469.

[Ship81] Shipman, D. W., "The Functional Data Model and the Data Language DAPLEX," ACM Transactions on Database Systems, Vol. 6, No. 1, March 1981, pp. 140–173.

[Ullm82] Ullman, J. D., "Principles of Database Systems," 2nd Edition, Computer Science Press, 1982.

[WY79] Wong, E., and K. Youssefi, "Decomposition — A Strategy for Query Processing," ACM Transactions on Database Systems, Vol. 1, No. 3, September 1976, pp. 223–241.