Institute for Logic, Language and Computation

THE MEANING OF DUPLICATES IN THE RELATIONAL DATABASE MODEL

Karen L. Kwast
Sieger van Denneheuvel

ILLC Prepublication Series
for Computation and Complexity Theory CT-93-04

University of Amsterdam

Institute for Logic, Language and Computation
Plantage Muidergracht 24
1018TV Amsterdam
Telephone 020-525.6051, Fax: 020-525.5101

THE MEANING OF DUPLICATES IN THE RELATIONAL DATABASE MODEL

Karen L. Kwast
Sieger van Denneheuvel
Department of Philosophy
University of Amsterdam

ILLC Prepublications
for Computation and Complexity Theory
ISSN 0928-3323

Coordinating editor: Dick de Jongh

received May 1993

The ILLC Prepublication Series

1990

Logic, Semantics and Philosophy of Language

LP-90-01 Jaap van der Does	A Generalized Quantifier Logic for Naked Infinitives
LP-90-02 Jeroen Groenendijk, Martin Stokhof	Dynamic Montague Grammar
LP-90-03 Renate Bartsch	Concept Formation and Concept Composition
LP-90-04 Aarne Ranta	Intuitionistic Categorial Grammar
LP-90-05 Patrick Blackburn	Nominal Tense Logic
LP-90-06 Gennaro Chierchia	The Variablity of Impersonal Subjects
LP-90-07 Gennaro Chierchia	Anaphora and Dynamic Logic
LP-90-08 Herman Hendriks	Flexible Montague Grammar
LP-90-09 Paul Dekker	The Scope of Negation in Discourse, towards a Flexible Dynamic Montague grammar
LP-90-10 Theo M.V. Janssen	Models for Discourse Markers
LP-90-11 Johan van Benthem	General Dynamics
LP-90-12 Serge Lapierre	A Functional Partial Semantics for Intensional Logic
LP-90-13 Zhisheng Huang	Logics for Belief Dependence
LP-90-14 Jeroen Groenendijk, Martin Stokhof	Two Theories of Dynamic Semantics
LP-90-15 Maarten de Rijke	The Modal Logic of Inequality
LP-90-16 Zhisheng Huang, Karen Kwast	Awareness, Negation and Logical Omniscience
LP-90-17 Paul Dekker	Existential Disclosure, Implicit Arguments in Dynamic Semantics

Mathematical Logic and Foundations

ML-90-01 Harold Schellinx	Isomorphisms and Non-Isomorphisms of Graph Models
ML-90-02 Jaap van Oosten	A Semantical Proof of De Jongh's Theorem
ML-90-03 Yde Venema	Relational Games
ML-90-04 Maarten de Rijke	Unary Interpretability Logic
ML-90-05 Domenico Zambella	Sequences with Simple Initial Segments
ML-90-06 Jaap van Oosten	Extension of Lifschitz' Realizability to Higher Order Arithmetic, and a Solution to a Problem of F. Richman
ML-90-07 Maarten de Rijke	A Note on the Interpretability Logic of Finitely Axiomatized Theories
ML-90-08 Harold Schellinx	Some Syntactical Observations on Linear Logic
ML-90-09 Dick de Jongh, Duccio Pianigiani	Solution of a Problem of David Guaspari
ML-90-10 Michiel van Lambalgen	Randomness in Set Theory
ML-90-11 Paul C. Gilmore	The Consistency of an Extended NaDSet

Computation and Complexity Theory

CT-90-01 John Tromp, Peter van Emde Boas	Associative Storage Modification Machines
CT-90-02 Sieger van Denneheuvel, Gerard R. Renardel de Lavalette	A Normal Form for PCSJ Expressions
CT-90-03 Ricard Gavaldà, Leen Torenvliet, Osamu Watanabe, José L. Balcázar	Generalized Kolmogorov Complexity in Relativized Separations
CT-90-04 Harry Buhrman, Edith Spaan, Leen Torenvliet	Bounded Reductions
CT-90-05 Sieger van Denneheuvel, Karen Kwast	Efficient Normalization of Database and Constraint Expressions
CT-90-06 Michiel Smid, Peter van Emde Boas	Dynamic Data Structures on Multiple Storage Media, a Tutorial
CT-90-07 Kees Doets	Greatest Fixed Points of Logic Programs
CT-90-08 Fred de Geus, Ernest Rotterdam, Sieger van Denneheuvel, Peter van Emde Boas	Physiological Modelling using RL
CT-90-09 Roel de Vrijer	Unique Normal Forms for Combinatory Logic with Parallel Conditional, a case study in conditional rewriting

Other Prepublications

X-90-01 A.S. Troelstra	Remarks on Intuitionism and the Philosophy of Mathematics, Revised Version
X-90-02 Maarten de Rijke	Some Chapters on Interpretability Logic
X-90-03 L.D. Beklemishev	On the Complexity of Arithmetical Interpretations of Modal Formulae
X-90-04	Annual Report 1989
X-90-05 Valentin Shehtman	Derived Sets in Euclidean Spaces and Modal Logic
X-90-06 Valentin Goranko, Solomon Passy	Using the Universal Modality: Gains and Questions
X-90-07 V.Yu. Shavrukov	The Lindenbaum Fixed Point Algebra is Undecidable
X-90-08 L.D. Beklemishev	Provability Logics for Natural Turing Progressions of Arithmetical Theories
X-90-09 V.Yu. Shavrukov	On Rosser's Provability Predicate
X-90-10 Sieger van Denneheuvel, Peter van Emde Boas	An Overview of the Rule Language RL/1
X-90-11 Alessandra Carbone	Provable Fixed points in $I\Delta_0+\Omega_1$, revised version
X-90-12 Maarten de Rijke	Bi-Unary Interpretability Logic
X-90-13 K.N. Ignatiev	Dzhaparidze's Polymodal Logic: Arithmetical Completeness, Fixed Point Property, Craig's Property
X-90-14 L.A. Chagrova	Undecidable Problems in Correspondence Theory
X-90-15 A.S. Troelstra	Lectures on Linear Logic

1991

Logic, Semantics and Philosophy of Langauge

LP-91-01 Wiebe van der Hoek, Maarten de Rijke	Generalized Quantifiers and Modal Logic
LP-91-02 Frank Veltman	Defaults in Update Semantics
LP-91-03 Willem Groeneveld	Dynamic Semantics and Circular Propositions
LP-91-04 Makoto Kanazawa	The Lambek Calculus enriched with Additional Connectives
LP-91-05 Zhisheng Huang, Peter van Emde Boas	The Schoenmakers Paradox: Its Solution in a Belief Dependence Framework
LP-91-06 Zhisheng Huang, Peter van Emde Boas	Belief Dependence, Revision and Persistence
LP-91-07 Henk Verkuyl, Jaap van der Does	The Semantics of Plural Noun Phrases
LP-91-08 Víctor Sánchez Valencia	Categorial Grammar and Natural Reasoning
LP-91-09 Arthur Nieuwendijk	Semantics and Comparative Logic
LP-91-10 Johan van Benthem	Logic and the Flow of Information

Mathematical Logic and Foundations

ML-91-01 Yde Venema	Cylindric Modal Logic
ML-91-02 Alessandro Berarducci, Rineke Verbrugge	On the Metamathematics of Weak Theories
ML-91-03 Domenico Zambella	On the Proofs of Arithmetical Completeness for Interpretability Logic
ML-91-04 Raymond Hoofman, Harold Schellinx	Collapsing Graph Models by Preorders
ML-91-05 A.S. Troelstra	History of Constructivism in the Twentieth Century
ML-91-06 Inge Bethke	Finite Type Structures within Combinatory Algebras
ML-91-07 Yde Venema	Modal Derivation Rules
ML-91-08 Inge Bethke	Going Stable in Graph Models
ML-91-09 V.Yu. Shavrukov	A Note on the Diagonalizable Algebras of PA and ZF
ML-91-10 Maarten de Rijke, Yde Venema	Sahlqvist's Theorem for Boolean Algebras with Operators
ML-91-11 Rineke Verbrugge	Feasible Interpretability
ML-91-12 Johan van Benthem	Modal Frame Classes, revisited

Computation and Complexity Theory

CT-91-01 Ming Li, Paul M.B. Vitányi	Kolmogorov Complexity Arguments in Combinatorics
CT-91-02 Ming Li, John Tromp, Paul M.B. Vitányi	How to Share Concurrent Wait-Free Variables
CT-91-03 Ming Li, Paul M.B. Vitányi	Average Case Complexity under the Universal Distribution Equals Worst Case Complexity
CT-91-04 Sieger van Denneheuvel, Karen Kwast	Weak Equivalence
CT-91-05 Sieger van Denneheuvel, Karen Kwast	Weak Equivalence for Constraint Sets
CT-91-06 Edith Spaan	Census Techniques on Relativized Space Classes
CT-91-07 Karen L. Kwast	The Incomplete Database
CT-91-08 Kees Doets	Levationis Laus
CT-91-09 Ming Li, Paul M.B. Vitányi	Combinatorial Properties of Finite Sequences with high Kolmogorov Complexity
CT-91-10 John Tromp, Paul Vitányi	A Randomized Algorithm for Two-Process Wait-Free Test-and-Set
CT-91-11 Lane A. Hemachandra, Edith Spaan	Quasi-Injective Reductions
CT-91-12 Krzysztof R. Apt, Dino Pedreschi	Reasoning about Termination of Prolog Programs

The meaning of duplicates in the relational database model

K.L. Kwast & S.J. van Denneheuvel*
Institute for Logic, Language and Computation
Dept of Philosophy, University of Amsterdam
Nieuwe Doelenstraat 15, 1012 CP, Amsterdam, NL
kwast@illc.uva.nl

April 1993

Abstract

The relational database model is defined in terms of sets, whereas SQL needs the `DISTINCT` option for explicit duplicate removal. We define the underlying concept of duplicate tuples, generalize the operators of the relational algebra and study the connection with logic. It is shown that 'baggy' operators are neither classical nor linear. The generalized system is applied in a formal interpretation of SQL.

Keywords. Duplicates; relational databases; relational algebra; classical logic; linear logic; SQL.

Contents

*present address: Sy*ll*ogic, postbus 26, 3990 DA, Houten, NL.

1 Introduction

According to the relational database model, all information is stored in relations, that is, *sets* of tuples. This has the advantage of being a well-defined and well-known mathematical notion. However, in current implementations of the relational database model (: SQL), it is customary to allow for duplicate tuples in the result of a relational query. It may be assumed that the procedural meaning of these duplicates is clear to all people that make use of such an implementation. Still, there is no obvious interpretation of duplicate tuples: sets just don't have duplicates. Hence it is time to investigate whether or not there exists a mathematical notion of *bags* (: collections of tuples that may have duplicates).

1.1 Lists and other datastructures

There is one type of objection to this project, namely that one can simply refer to elementary notions, such as the *list*, the *array*, or the *tree*, which are defined in introductory courses on datastructures. All basic operations on these structures are well-understood and tested by scores of implementations.

Nevertheless, it was one of the milestones of the relational model that the "naive user" was no longer required to deal with implementation issues. He was supposed to be allowed an extremely simple picture of how the database is organized: just a collection of flat tables, or 'relations', that satisfy comprehensible integrity constraints. The underlying datastructure was to be hidden from the user, letting him concentrate on *what* he wants, and leaving it up to the expert to decide *how* it is done, so as to guarantee that the right thing gets done in an efficient manner.

Hence, we need a simple and mathematically elegant notion to be used as an intermediate between theory and practice. If such a notion does not allow for duplicates, then Codd's model should not be polluted with duplicates. But if we can find a neat definition of so-called *bags*, then maybe we ought to change over to the *baggy database model.*

1.2 Some examples

Consider two friends that plan to have dinner together and are shopping in a local supermarket. Both are pushing a trolley, so as to be able to lean on it, and both collect groceries that might constitute a lovely meal. As it happens, there are tomatos in either trolley.

- How many tomatos do they have for dinner?

Suppose both friends have been inviting separately: Andy has talked to his collegues Chris, David, Emma and Fred, whereas Bob called on his friends Emma, Fred, George and Harry.

- How many people will come to dinner?
- Which of Andy's collegues are no friend of Bob?

Bob and Andy get rather bored waiting, so they play a game of naming animals with an 'E'. Pretty soon things get out of hand: Andy mentions 'elephants' twice; Bob notices his duplicity, but Andy persists, since the first were African elephants and the second Indian elephants.

- Is it possible to play 'naming animals' in this liberal manner?
- May Bob offer 'elephants', meaning those living in the London Zoo?
- Will 'eagle' and 'eaglet' count for one?

While cooking dinner, Andy and Bob discuss the local bus company. Bob estimates the number of passengers to be over 2 million a month. Andy disagrees, as there do not live that many people in the area.

- Can one individual be two passengers?
- Can one passenger be two individuals?

1.3 Identity criteria

Whether or not one individual can be two passengers could be the starting point of an intricate philosophical discussion on the nature of identity. Here, however, we are only concerned with identity as used in a database: when are 2 entities identical; can a database store duplicates?

Duplicates only make sense if there is an underlying property to distinguish between them. Nevertheless, this property should not be relevant for the identity criteria of the type of object, or else they would not be duplicates, but just objects sharing some properties.

In a database context there are three types of identity criteria:

- object identity
- value identity
- user-defined identity

Object identity is based on the unique identifier that each and every record in a database must have; different objects may coincide with respect to all other observable values. More often than not, object identity is based on physical unicity. All objects are identified by means of their unique location and the only way they can be referred to is by means of pointers. Related properties are linked to an object by means of a pointer to the object itself.

Value identity, on the other hand, conforms to Leibniz' law of the *identity of indiscernibles.* Value identity is a syntactic notion of equality. The connection between data requires that strings of characters or integers are compared, regardless of their physical location.

Finally, *user-defined identity* refers to the possibility to create for all items in a database primary key attributes; additional properties of these items and relations between them are stored with reference to the same key values. Examples are *personal identity numbers* (: `PIN`), *social security numbers* (: `SOFI`) and the like. A person is identified by means of these key attributes, either by identical values on `PIN`, or by linking his `PIN` to a unique `SOFI` number. Formulated in this manner user-defined identity presupposes value identity, but it can be based on object identity as well. In that case the objects are linked by means of an index or physical adjacency (: hierarchy of record types).

1.4 The system key SK

The relational database model represents information through value identity, at least as far as the user is concerned. Sorting a list of employees on `SOFI` number is vacuous from this point of view: only when the list is printed does the ordering become visible. Nevertheless, one can easily imagine that the computation of the join of 2 relations is greatly simplified if both are ordered on their common attributes. So sorting *is* relevant for the implementation of the database.

As long as we want to shield the user from implementation issues, we must stick to value identity. If duplicates (: identical tuples of values) are to be distinguished, it is by additional (albeit irrelevant) attribute values. In principle, duplicates will emerge from a *projection.* Since we cannot make use of the physical location of the identical tuples, and the removed attributes are 'no longer there', we must add some sort of *system key.* The additional attribute SK is known to exist, but impossible to access.

The value of a tuple on SK may be a pointer to its physical location (: on disk), or it may be a unique timestamp to witness the date of creation. This is, as far as I can see, irrelevant and up to the implementation. The only necessary requirement on SK is that it refers to a reserved data type: all comparisons between values on SK and any other attribute are meaningless and filtered away by well-formedness conditions.

The concept of a system key can be found in actual implementations of the relational database model, for instance in the Oracle SQL. From the users guide ([6], pg 197, 73):

> "Every table in the database has a column named ROWID. During a transaction, each row in a table has a fixed and unique ROWID value. [...] When a record is fetched into the workspace, the value of ROWID is fetched along with it. This enables some types of triggers to identify the table row corresponding to each record in the workspace. (Newly inserted records do not correspond to any row, and so have a null ROWID.) [...]
>
> Under no circumstances should an operator or trigger change the ROWID."

> "The address of every row in the database [...] can be retrieved in hexadecimal form by a SQL query using the reserved word ROWID. The information returned will locate the row by partition, by block, and by position of the row in the block. ROWIDs are stored in a logical -not physical- column, which sometimes can be accessed like table columns, i.e. used in `SELECT` or `WHERE` clauses, but:

- A ROWID is not guaranteed to be constant for any row over time. The physical location of a row may change due to updating, or exporting and reimporting.

- A ROWID is not stored in the database. It can be referenced like other data, but it is not actually a column of data. Therefore, you cannot `UPDATE`, `INSERT`, or `DELETE` a ROWID.

The usefulness of a ROWID lies in the fact that:

- It is the fastest means of accessing a specified row.
- It can be used to determine how many blocks of storage a table requires.
- It can be used to obtain row-level locks."

As can be seen from these quotes from the users guide, there is far more going on than the simple addition of a reserved attribute to the scheme. We will not analyse Oracle, even if it might make this issue a lot more practical and transparant. However, in a later section we consider the occurrences of duplicates in standard SQL queries and whether or not the inequivalences we encounter are relevant in real-life situations.

1.5 Outline of this report

The next section provides the basic definitions of relational database theory and a formalization of the notion of a bag. In § 3 the relational operators (: selection, projection, union and join) are adapted to the presence of duplicates. The join of 2 bags can be defined in several ways, depending on one's intuitions concerning the system keys. The straightforward generalization, treating SK as just another attribute, is rejected, as the result will be unstable: relocation affects the number of duplicates. Since the location is considered to be irrelevant, only used to get a meaningful definition of duplicates, and since the number of duplicates can be used to compute aggregates, such as: the average A-value, all operations ought to be stable on the number of duplicates. Alternative definitions are based on optimal relocations or else they are directed: the 'directed' join preserves the number of duplicates in the dominant bag.

Some additional operators (such as: sort) are mentioned in § 4. Sorting a relation is vacuous, but the effect can be seen in the result of a query, which will be printed in the specified order.

The number of alternative definitions is rather large, so an overview is given in § 5. The interpretation is characterized by the number of duplicates in a singleton interpretation: a bag of identical tuples.

Various equivalences, valid for relational operators, turn out to be unvalid for some combinations of baggy operators. A list of them is given in § 6. Relative to these equivalences the dependencies between the various options are studied (§ 7), aiming at an optimal combination, one that preserves as many equivalences as possible. There is no system of baggy operators that obey all rules of classical logic.

In § 8 the connection with linear logic is studied. We have found no simple translation from linear connectives to baggy operators, analoguous to the equivalence between classical connectives and relational operators. Linear logic distinguishes between *context-free* and *context-sensitive* connectives, but the baggy operators satisfy rules of both types simultaneously (: unlike the linear connectives). Hence the logic of duplicates cannot be equated with linear logic, either.

Finally, in § 9 the results are employed to analyse SQL.

2 Basic definitions

The notion of a system key can be formalized as follows. In [9] a database scheme was defined as a set of relation schemes, which specify the attributes of each relation (: its *scope*), plus a set of integrity constraints, such as funcional dependencies and primary- and foreign keys (cf. [13]). For simplicity sake, data types are ignored.

Definition 1 *Let* $\Sigma =< \alpha,\ \mathcal{X} >$ *be a database scheme for a language with relation names RELN and attributes* $\mathcal{A}$*, so* α *assigns a scope* $\alpha(R) \subset \mathcal{A}$ *to each relation name* R *and* $\mathcal{X}$ *is a finite set of constraints, then* $\Sigma^* =< \alpha^*,\ \mathcal{X}^* >$ *is an extended database scheme, with:*

$$\begin{aligned} \alpha^*(R) &:= \alpha(R) \cup \{\text{SK}\}, \\ \mathcal{X}^* &:= \mathcal{X} \cup \{R : \text{SK} \rightarrow A \mid A \in \alpha(R), R \in RELN\}. \end{aligned}$$

Hence, SK is in the extended scope of every relation and it is a primary key.

2.1 The database instance and its implementations

Once the database scheme is extended with a system key, the instance of a database must be adapted as well. Let us call the resulting instance an *implementation* of the original scheme.

For reasons of brevity we will use $\mathcal{N}$ (: the natural numbers) for SK and $\mathcal{D}$ (: the set of objects) for user-defined attributes, even if the latter will include integers. A leading @ for SK-values will be used to avoid confusion. *All* relations are extended with a column of unique system key values, but several relations may share key values (e.g.: derived relations). However, when the join of any set of relations is computed, the system key *must* remain a key. This means that the database can at any time be listed, implicitly ordered on SK, as a relation over all defined attributes, using inapplicability symbols (: $\perp$) to extend partial tuples. Let us call this the *compatibility requirement.*

Definition 2 *A database instance* $< \mathcal{D}, \mathcal{A}, \mathcal{I} >$ *of a relational scheme* Σ *has an implementation* $< \mathcal{N}, \mathcal{D}, \mathcal{A}, \mathcal{J} >$*, if:*

1. for each $R \in RELN(\Sigma)$: $\mathcal{J}(R)$ *is a set of functions* t*,*
such that $t(\text{SK}) \in \mathcal{N}$ *and* $t[\alpha(R)] \in \mathcal{I}(R)$.

2. the compatibility requirement is met:

$$\forall r \in \mathcal{J}(R),\ \forall s \in \mathcal{J}(S) :\ r =_{\text{SK}} s\ \Rightarrow\ r =_{\alpha(R)\cap\alpha(S)} s,$$
that is, if $r(\text{SK}) = s(\text{SK})$*, then* $r(A) = s(A)$ *for all* $A \in \alpha(R) \cap \alpha(S)$.

The compatibility requirement guarantees that all tuples with identical SK-value are restrictions of a single *universal* tuple with that SK-value. Note that we do *not* make the universal relation assumption (cf. [13]), since all relations may have disjoint sets of SK-values; the compatibility requirement reflects the intuitive interpretation of the system key as pointers in an index.

It will be assumed that *all* tuples are extended at *all* times with a value on SK, let's say to indicate that the intermediate result is written down (temporarily) on this location / at this moment. For instance, when evaluating a query, each step in the computation (: the result of each relational operation on the given relations) is written down as an extended relation.

Since the user is not at all interested in the value of SK, we want to ignore this attribute whenever we can. So we will use our standard notation without reference to SK. (This choice may be confusing; however, the standard notation must be adapted for either the instance (without SK) or the implementation (with SK). Since we do not conceive the implementation as a standard instance, this is the one that has to adjust.) The notation will be explained by means of an example.

Consider a tuple $t = < a, b, c >$ and a relation $\mathcal{I}(R) = \{< a, b, c >, < d, e, f >\}$. Both are extended with SK-values, but we will say that $t \in \mathcal{I}(R)$ on the basis of the 3 user-defined attribute values. If in addition the SK-values match, then t is (pointing to) a tuple in the implemented relation $\mathcal{J}(R)$.

We will write $t \in_i R$ to indicate that the extended tuple is in the extended relation. Compare:

$$\begin{array}{lll} < @123, a, b, c > & \in_i & \{< @123, a, b, c >, < @456, d, e, f >\} \\ < @789, a, b, c > & \notin_i & \{< @123, a, b, c >, < @456, d, e, f >\} \\ < @789, a, b, c > & \in & \{< @123, a, b, c >, < @456, d, e, f >\} \end{array}$$

Obviously, $R \subseteq_i S$ is defined in terms of $\in_i$ as usual (: $\forall t \in_i R : t \in_i S$). Likewise, $t =_i r$ iff t and r are identical, SK included and $t \neq_i r$ iff the tuples disagree on some attribute, possibly SK. As a consequence, the standard notions become slightly confusing, at least when the extension is not completely ignored.

Lemma 1 *For all extended tuples and relations:*

$$\begin{array}{lll} t = r & \textit{iff} & \forall A \in \mathcal{A} : t(A) = r(A) \\ t =_i r & \textit{iff} & t = r \ \& \ t(\text{SK}) = r(\text{SK}) \\ t \neq r & \textit{iff} & \exists A \in \mathcal{A} : t(A) \neq r(A) \\ t \neq_i r & \textit{iff} & t \neq r \ \textit{ or } t(\text{SK}) \neq r(\text{SK}) \\ t \in R & \textit{iff} & \exists r \in_i R : r = t \end{array}$$

Finally, we can define duplicates as identical tuples with distinct locations:

Definition 3 *Tuples r and t are* **duplicates** *iff $r = t$ and $r(\text{SK}) \neq t(\text{SK})$.*

Note that duplicates are indeed distinguished by a totally irrelevant property: no user will have any interest in the precise location of a stored tuple.

2.2 Bags as multisets

There is an alternative concept that can be used to explain duplicates in relations: multisets. Instead of the standard interpretation of a relation as a set of tuples on a fixed scope, one can employ finite sequences closed under permutation (cf. [12]). The mathematical concept requires an additional attribute, *multiplicity*, that denotes the number of duplicates. A set is a special case of a multiset, namely one in which all tuples have multiplicity 1.

We will not adopt this approach, since it has no explanatory power; the intrinsic nature of duplicates remains unclear. Moreover, a concept such as 'sorting of a multiset' is meaningless. Nevertheless, in the following pages we will argue that the generalized relational operations on bags should be homomorphous over relocation. This will ensure that only the multiplicity of duplicates is relevant, so our implementation will be functionally equivalent to one based on multisets (see § 5).

2.3 Location operators

In this section we introduce a global and a local relocation operator, to arbitrarily change the values on SK and a set operator, which removes duplicates. These operators are all meaningless on relations (: sets).

The internal system key SK has no declarative content, so the result of a query should not be affected by changes in the actual SK-values. Take, for instance, a selection query such as:

$\sigma_{\mathrm{SK}=@123}(R)$ `SELECT * FROM R WHERE SK = @123;`

This is not the type of query a naive user should be trusted with.

In order to ensure that the actual SK-values are irrelevant, all baggy operators should be homomorphous over *global relocation*.

Definition 4 *Let* ρ : SK $\rightarrow$ SK *be a permutation of* $\mathcal{N}$, *then* $< \mathcal{N}, \mathcal{D}, \mathcal{A}, \rho \circ \mathcal{J} >$ *is a global relocation of the implementation* $< \mathcal{N}, \mathcal{D}, \mathcal{A}, \mathcal{J} >$, *consisting of replaced bags*

$$\rho \circ \mathcal{J}(R) := \{ t \mid \exists r \in_i \mathcal{J}(R) : t = r \ \&\ t(\mathrm{SK}) = \rho(r(\mathrm{SK})) \}$$

Global relocation is a non-deterministic operator that preserves the compatibility requirement. One can think of DBMS routines, such as: garbage collection, implementation of a new index to improve response time or even the physical side-effects of query processing, where data is transferred from secondary to main memory and vice versa.

Hence we have the following constraint on baggy operators: the result of applying an operator f to bags R and S may not be affected by global relocation (except that the result is relocated, along with its arguments). Formally, f is homomorphous over ρ: $\rho(f(R,S)) = f(\rho(R), \rho(S))$.

The following constraint on baggy operators is more controversial: baggy operators should be homomorphous over *local relocation*. This reflects the philosophy of the relational database model that *value identity* is more user-friendly than *object identity*. In particular, it should be irrelevant to the outcome of a query -though it will of course affect the performance of the system- whether or not the database is normalized. This means that queries on *views* should be interchangeable with queries on the underlying relations through substitution of the view definition.

For instance, if $V := \Pi_X(R)$, then $\sigma_\phi(V) \equiv \sigma_\phi\Pi_X(R)$ (: the result of both queries is identical, regardless of the content of the database). Now the main effect of defining a view is that the result can be stored elsewhere. So the SK-values of V may differ from R's, which implies that they cannot be relevant to the outcome of the query. This requires stability under local relocation.

The relocation operator † is a subsitution on SK in a single relation, a non-deterministic operation that must preserve the compatibility requirement. It yields either the relation itself or a copy.

Definition 5 $R\dagger := \{t \mid \exists r \in_i R: \ t = r \text{ and } t(\text{SK}) = f(r(\text{SK}))\}$ *for some bijection f on $\mathcal{N}$, such that the result is compatible with all other implemented relations.*

Note the traditional sloppiness to write R instead of $\mathcal{J}(R)$ and $R\dagger$ instead of $\mathcal{J}(R\dagger)$ or $\mathcal{J}(R)\dagger$.

We will write $S = R\dagger$ if S can be obtained from R by relocation. Similarly, $S \neq R\dagger$ means that there is no possibility to obtain S by relocation of R.

Example 1 *Some relocations of R.*

R	SK	A
	@1	a
	@2	b
	@3	b

$R\dagger$	SK	A
	@4	a
	@5	b
	@6	b

$R\dagger$	SK	A
	@1	a
	@4	b
	@7	b

Obviously, relocation does not change the number of duplicates, so $R \simeq R\dagger$ (: there is a bijection). In particular, the absence or presence of tuples is not affected: both $R \subseteq R\dagger$ and $R\dagger \subseteq R$, since subsets ignore the SK-values.

Local relocation can be part of the query processing: in order to compute a query, the relevant relations could be copied from secondary to main memory, where all intermediate results are written down as well.

There are no constraints on local relocation concerning matching tuples: 2 tuples from different relations may get or loose an identical SK-value as a result of relocation. For instance, if R and S are views on a common relation T, then relocation may 'remove' R-tuples from their corresponding S-tuples and matching tuples may be 'identified'.

Example 2 *Let $R := \Pi_{AB}(T)$ and $S := \Pi_{BC}(T)$ be 2 views on T. Possible implementations:*

T	SK	A	B	C
	@1	a	b	c
	@2	e	b	g

$R\dagger$	SK	A	B
	@3	a	b
	@4	e	b

$S\dagger$	SK	B	C
	@4	b	c
	@5	b	g

The value oriented approach of the relational database model requires that queries on these views are evaluated without any concern for the actual SK-values. So, in order to ensure that all possible relocations of an relation R behave the same, we must require of any baggy operator f that it is homomorphous over †: $f(R, S) = f(R\dagger, S\dagger)\dagger$

This requirement is not met by arbitrary operators on bags (see § 3.3 below). If not, then the least that must hold is that $\lceil f(R,S)\rceil = \lceil f\rceil(\lceil R\rceil, \lceil S\rceil)$, where $\lceil f\rceil$ is the standard version of the baggy operator f and $\lceil T\rceil$ is the set of tuples in T.

The set operator $\lceil - \rceil$ removes duplicates by deleting SK from the set of attributes:

Definition 6 $\lceil R\rceil \;:=\; \{t : \alpha(R) \mid \exists r \in R :\; r = t\}$.

Obviously, we must locate the resulting set by means of some location operator. If R is any set of tuples, then R^* is an implementation of R, a bag with unique tuples. One can define R^* as the alphabetically ordered list of R-tuples extended with increasing free SK-values, but it is really up to the DBMS, that is, beyond the interest of the naive user. The location operator * is again non-deterministic and preserves the compatibility requirement.

The result of $\lceil R\rceil^*$ is that all duplicates are removed from R, comparable to the `DISTINCT` option from SQL. An alternative method of duplicate removal is to put all possible SK-values in $t(\text{SK})$ (: $t(\text{SK}) = *$) for all $t \in \lceil R\rceil$, but this would would violate the compatibility requirement.

In the sequel, we will not distinguish between the set $\lceil R\rceil$ and the duplicate-free bag $\lceil R\rceil^*$ on an arbitrary location.

3 Relational operators

In this section we will adapt all relational operators to work on bags. Some definitions will yield identical results, regardless of the content of the database. This can be expressed formally by means of the concept of term equivalence:

Definition 7 $R \equiv S,\ R \not\equiv S$.
Two terms R and S over a scheme Σ are equivalent iff for every implementation $\mathcal{J}$ of Σ: $\mathcal{J}(R) = \mathcal{J}(S)\dagger$.
Two terms R and S are inequivalent iff there exists some implementation $\mathcal{J}$ such that $\mathcal{J}(R) \neq \mathcal{J}(S)\dagger$.

For instance, $R \equiv R\dagger$, but $R \not\equiv \lceil R\rceil$, since the number of duplicates is reduced by duplicate removal in any implementation that has duplicates in R. Term equivalence will be treated in detail in § 6.

3.1 Projection

What can be simpler than projection? All that has to be done is remove some columns.

That may be, but if the result of a projection must be a relation, then one has to remove duplicates from the implemented list.

Definition 8 $\Pi_X(R) \;:=\; \{t : X \mid \exists r \in R :\; t =_X r\}^*$

In this case the projection may or may not contain new values on SK (: the result can be located elsewhere).

Duplicates are retained when identical subtuples are distinguished by their SK-values.

Definition 9 $\Pi_{iX}(R) \ := \ \{t : X^* \mid \exists r \in_i R : \ t =_{X,\mathrm{SK}} r\}$

In this definition the projection is located in the original relation itself, either physically or by means of pointers or identical timestamps. This may or may not be desirable; a third possibility is to relocate the projection. The resulting baggy projection can be defined in terms of Π_i and the relocation operator $\dagger$, as follows:

Definition 10 $\pi_X(R) \ := \ (\Pi_{iX}(R))\dagger$

Since we do not consider the exact location of any remote interest, we will from now on ignore Π_i in favour of π, except when the actual implementation is concerned. The difference between Π and π is illustrated in the following (trivial) example.

Example 3 *Projection with and without duplicate removal.*

R	A	B	C	D
@1	a	b	c	d
@2	e	b	c	f
@3	g	h	c	i
@4	j	k	l	m

$\Pi_{BC}(R)$	B	C
@11	b	c
@12	h	c
@13	k	l

$\pi_{BC}(R)$	B	C
@21	b	c
@22	b	c
@23	h	c
@24	k	l

The relational projection Π can be derived from baggy projection π by means of the set operator $\lceil - \rceil$ in the obvious manner, which shows that the former is a suitable generalization of the latter in the context of implemented instances.

Lemma 2 *For any bag R:*
1. $\Pi_X(R) \ \equiv \ \lceil \pi_X(R) \rceil$
2. $\Pi_X(R) \ \not\equiv \ \pi_X(\lceil R \rceil)$

Baggy projection can create bags even if its input is a relation. In fact, projection is the main source of duplicates in **SQL**.

3.2 Selection

The interpretation of a selection expression is trivially generalized to deal with bags. We will not introduce a separate symbol to denote selections on bags, as there is no possibility of confusion: the number of duplicates is not affected for any tuple that satisfies the selection condition.

Definition 11 $\sigma_\phi(R) \ := \ \{r \in_i R \mid r \models \phi\}\dagger$

On account of the global relocation constraint, it is not allowed to have SK in the selection clause; the query $\sigma_{\mathrm{SK}=@123}(R)$ is meaningless.

3.3 Union

In taking a union one can either ignore SK or retain duplicates. As usual, the operator is only defined on compatible relations (: relations with identical scopes).

Definition 12 $R \cup S \ := \ \{t : \alpha(R) \mid \exists r \in R : \ r = t \ or \ \exists s \in S : \ s = t\}^*$

This definition is a proper generalization of the standard relational operator. When duplicates are retained, there is still an option as to how many of these duplicates are saved. If relocation is considered to be a meaningful operation, one may favour the following definition.

Definition 13 $R\cup_i S \;:=\; \{t : \alpha^*(R) \mid \exists r\in_i R :\; r{=_i}t \textit{ or } \exists s\in_i S :\; s{=_i}t\}$

Unfortunately, there is no intuitive interpretation of relational union that corresponds with this definition. Given 2 baggy relations R and S, if R contains 2 duplicates of some tuple t and S contains 3 duplicates of the same tuple, then the union$_i$ can have anything between 3 to 5 tuples.

Example 4 *Location dependent union.*

	$R\cup_i S$	A	B
	@1	*a*	*b*
R {	*@2*	*c*	*d*
	@3	*c*	*d*
	@4	*c*	*d*
	@5	*c*	*d*
S {	*@6*	*c*	*d*
	@7	*e*	*f*
	@8	*g*	*h*

	$R\dagger\cup_i S\dagger$	A	B
	@11	*a*	*b*
R †{	*@12*	*c*	*d*
	@13	*c*	*d*
	@14	*c*	*d*
S †{	*@15*	*c*	*d*
	@16	*e*	*f*
	@17	*g*	*h*

	$R\dagger\cup_i S\dagger$	A	B
	@21	*a*	*b*
R †{	*@22*	*c*	*d*
	@23	*c*	*d*
S †{	*@24*	*c*	*d*
	@25	*e*	*f*
	@26	*g*	*h*

The relations R and S are distinct, that is, the sets of their sk-values are disjoint. After relocation they may share a single duplicate (: $< @13, c, d >$) or even all R's $< c, d >$ duplicates (: @22, @23). Example 4 is conclusive; the operator $\cup_i$ is not homomorphous over relocation:

$$(R\cup_i S)\dagger \;\not\equiv\; (R\dagger\cup_i\, S\dagger).$$

As a consequence, we cannot freely substitute views for relations in an internal union. Let $R := S$ be a copy with distinct SK-values. Then, even though $R \equiv S$, still:

$$R \equiv R\cup_i R \not\equiv R\cup_i S \not\equiv S\cup_i S \equiv S.$$

The local relocation requirement is violated by all extended binary operators (: $\cap_i$, $\bowtie_i$, $\setminus_i$), which will be defined only as a tool to describe the system. (These operators all satisfy the global relocation requirement, that is, they do not depend on the actual SK-values.)

Still, before we dismiss these *internal operators* completely, there exist strong arguments in favour of them. For instance, query optimization for complex queries may require the internal operators. As an example, suppose $\alpha(\varphi \vee \psi) \cap X = \emptyset$, then

$$\Pi_X\sigma_\varphi(R) \cup \Pi_X\sigma_\psi(R) \;\equiv\; \Pi_X\sigma_{\varphi\vee\psi}(R).$$

This equivalence is preserved when R is implemented and all operators are replaced by their internal generalization. All duplicates in $\Pi_{iX}\sigma_\varphi(R)$ may or may not come from the same source as those in $\Pi_{iX}\sigma_\psi(R)$, so the former could have 2 duplicates, the latter 3 and the union anything between 3 to 5:

Example 5 *Internal union in a complex term.*

R	*A*	*B*	*C*
@1	c	*d*	*1*
@2	c	*d*	*2*
@3	c	*d*	*3*
@4	c	*d*	*4*
@5	c	*d*	*5*

$$\Pi_{iAB}\sigma_{C\leq 2}(R)\cup_i\Pi_{iAB}\sigma_{C\leq 3}(R) \equiv \Pi_{iAB}\sigma_{C\leq 3}(R)$$

$$\Pi_{iAB}\sigma_{2\leq C\leq 3}(R)\cup_i\Pi_{iAB}\sigma_{3\leq C\leq 5}(R) \equiv \Pi_{iAB}\sigma_{2<C<5}(R)$$

$$\Pi_{iAB}\sigma_{C\leq 2}(R)\cup_i\Pi_{iAB}\sigma_{C\geq 3}(R) \equiv \Pi_{iAB}(R)$$

It is easy to see that if the union-subterms are relocated before taking the union, then the 'source' of each tuple is lost and one cannot distinguish between these 3 queries, so all will be answered by the same number of duplicates. As a consequence, the equivalences given are no longer valid and one must ask the right-hand side queries to get the correct result.

If $\cup_i$ is not a location-independent operator, we must find alternatives. A sensible definition of union will retain as few duplicates as possible (: 3, in the example above), or else, as many as possible (: 5). Reasons to prefer one definition to the other will depend on the intended application: if the arguments are derived from a common source, then not all duplicates are new, but if the arguments are unrelated, then all duplicates should be saved. To start with the former, the *minimal union*:

Definition 14 $R\sqcup S := R\dagger\cup_i S\dagger$,
for relocation $R\dagger$ *and* $S\dagger$ *such that* $\forall r\in_i R\dagger, s\in_i S\dagger : r = s \Rightarrow r\in_i S\dagger$ or $s\in_i R\dagger$.

This definition is not very elegant, but then neither is the defined concept. It asks for the optimal choice of relocation that makes $R\dagger\cup_i S\dagger$ as small as possible. Let us call this a *relocation with maximal overlap.*

To get the largest possible bag of tuples, use the *maximal union*:

Definition 15 $R\uplus S := R\dagger\cup_i S\dagger$,
for relocation $R\dagger$ *and* $S\dagger$ *such that* $\forall r\in_i R\dagger, s\in_i S\dagger : r\neq_i s$.

The condition on the relocation of R and S in this definition can be formulated in terms of their sets of system keys: $R\dagger[\text{SK}] \cap S\dagger[\text{SK}] = \emptyset$.

An example will illustrate the difference between $\uplus$ and $\sqcup$.

Example 6 *Minimal and maximal union.*

R	*A*	*B*
@1	*a*	*b*
@2	c	*d*
@3	c	*d*

S	*A*	*B*
@4	c	*d*
@5	c	*d*
@6	c	*d*
@7	*e*	*f*
@8	*g*	*h*

$R\sqcup S$	*A*	*B*
@10	*a*	*b*
@24	c	*d*
@35	c	*d*
@06	c	*d*
@07	*e*	*f*
@08	*g*	*h*

$R\uplus S$	*A*	*B*
@1	*a*	*b*
@2	c	*d*
@3	c	*d*
@4	c	*d*
@5	c	*d*
@6	c	*d*
@7	*e*	*f*
@8	*g*	*h*

As desired, the standard definition can be derived from all others by means of duplicate removal:

Lemma 3 $\lceil R \rceil \cup \lceil S \rceil \;\equiv\; \lceil (R {\cup_i} S) \rceil \;\equiv\; \lceil (R \sqcup S) \rceil \;\equiv\; \lceil (R \uplus S) \rceil$

There is no connection between $\sqcup$ and $\uplus$, except in the trivial case that $R \cap S = \emptyset$.

The special case of $R \cup R$ deserves some additional attention. Since $\sqcup$ requires maximal overlap, $R \equiv R \sqcup R$, but $\uplus$ requires no overlap between the SK-values at all, so $R \uplus R = R\dagger{\cup_i}R\dagger$ for 2 distinct copies of R, yielding the doubling of R. It is no serious alternative to 'co-ordinate' the relocations, in order to preserve $R \uplus R \equiv R$, since that would ruin free view substitution: if $V := R$, then $V \equiv R$ but $R \uplus V \not\equiv R \uplus R$. In our value oriented approach we prefer to preserve $R \uplus V \equiv R \uplus R$ in favour of $R \uplus R \equiv R$.

There may be even more versions of union worthy of consideration. For instance, we could take the average number of duplicates (instead of the maximum or minimum). This one does not lead to a smooth operator, say $\oplus$: it validates $R \equiv R \oplus R$, but it is not even associative: $(R \oplus S) \oplus T \;\not\equiv\; R \oplus (S \oplus T)$. More importantly, it is hard to think of a plausible interpretation of the average union.

Before we turn to intersection, just a remark on the practical relevance of the various definitions. The maximal union can be found in SQL: `UNION ALL` corresponds with $\uplus$. (Indeed, `SELECT * FROM R UNION ALL SELECT * FROM R` is not the same query as `SELECT * FROM R`.) The standard operator $\cup$ is simply `UNION`. The minimal union $\sqcup$ corresponds to `OR` (: in selection conditions), though it will only be invoked in situations where both subterms have either 0 or the same number of duplicates.

3.4 Intersection

For the intersection of 2 bags of tuples there is likewise a choice of definitions. To ensure that relations yield a relation, let $\cap$ be the 'standard' definition.

Definition 16 $R \cap S \;:=\; \{t : \alpha(R) \mid \exists r \in R :\; r = t \textit{ and } \exists s \in S :\; s = t\}^*$

This definition is a proper generalization of the standard relational operator. When duplicates are present, one can, as before, treat SK as a genuine attribute, though the result will vary under relocation.

Definition 17 $R{\cap_i}S \;:=\; \{t \mid \exists r{\in_i}R :\; r{=_i}t \textit{ and } \exists s{\in_i}S :\; s{=_i}t\}$

This definition can only make sense when speaking over the baggy model, but it is unsuitable as a query operator: the internal intersection can be empty even if both bags have several duplicates of the same tuple, namely if these duplicates are located on different places (: unequal SK-values).

There is no point in aiming at the minimal intersection: it is empty, regardless of the operands. The *maximal intersection* tries to match as many tuples as it possibly can.

Definition 18 $R \sqcap S \;:=\; R\dagger{\cap_i}S\dagger$,
for relocations with maximal overlap: $\forall r{\in_i}R\dagger, s{\in_i}S\dagger :\; r = s \Rightarrow r{\in_i}S\dagger$ or $s{\in_i}R\dagger$.

Finally, one may opt to retain the number of duplicates in R (: the left-hand side of the intersection operator), thereby ruining commutativity. This reduces intersection to selection, asking for those R-tuples that have copies in S.

Definition 19 $R\vec{\cap}S \;:=\; \{t \mid t\in_i R \textit{ and } t \in S\}\dagger$.

This type of intersection will be called *directed.* Note that there is no directed version of union -it makes no sense.

Obviously, if R is a relation then $R\vec{\cap}S \;=\; R \cap S \;=\; R \sqcap S$. Moreover:

Lemma 4 $\lceil R\rceil \cap \lceil S\rceil \;\equiv\; \lceil (R \sqcap S)\rceil \;\equiv\; \lceil (R\vec{\cap}S)\rceil$
However, $\lceil R\rceil \cap \lceil S\rceil \;\not\equiv\; \lceil (R\cap_i S)\rceil$.

The inequivalence means that $\cap_i$ is *not* a reasonable generalization of $\cap$, since duplicate removal *after* taking the intersection could be too late.

Example 7 *Maximal and directed intersection.*

R	A	B
@1	*a*	*b*
@2	*c*	*d*
@3	*c*	*d*
@4	*c*	*d*

S	A	B
@5	*c*	*d*
@6	*c*	*d*
@7	*e*	*f*
@8	*e*	*f*

$R \cap S$	A	B
@11	*c*	*d*

$R\cap_i S$	A	B
-		

$R \sqcap S$	A	B
@11	*c*	*d*
@12	*c*	*d*

$R\vec{\cap}S$	A	B
@22	*c*	*d*
@23	*c*	*d*
@24	*c*	*d*

$S\vec{\cap}R$	A	B
@35	*c*	*d*
@36	*c*	*d*

3.5 Join

The various types of baggy intersection can be generalized to deal with incompatible relations, that is, to the join of 2 bags of tuples.

Definition 20 $R \bowtie S \;:=\; \{t : \alpha(R) \cup \alpha(S) \mid \exists r \in R :\; r =_X t \textit{ and } \exists s \in S :\; s =_X t\}^*$,
where $X = \alpha(R) \cap \alpha(S)$.

The standard definition can be applied on the implementation directly:

Definition 21 $R\bowtie_i S \;:=\; \{t \mid \exists r\in_i R :\; r =_X t \textit{ and } \exists s\in_i S :\; s =_X t\}$
where $X = \alpha^*(R) \cap \alpha^*(S)$ *(:* SK $\in X$*).*

Note that the result cannot be larger than the smallest argument.

The join with duplicates should join all matching tuples and keep as many duplicates as possible, namely the minimal number of duplicates in a matching pair of tuples. Its formulation is no trivial matter. Consider the obvious generalization of $\sqcap$:

Definition 22 $R \sqcap S \;:=\; R\dagger\bowtie_i S\dagger$,
for relocations $R\dagger$ *and* $S\dagger$ *with maximal overlap on* X*:*
$\forall r\in_i R\dagger, s\in_i S\dagger :\; r =_X s \Rightarrow (\; r[\textsc{sk}] \in S\dagger[\textsc{sk}]$ or $s[\textsc{sk}] \in R\dagger[\textsc{sk}]\;)$.

This definition does not yield a join, as can be seen from the following example:

Example 8 *3 joins by maximal overlap.*

R	A	B
@1	*a*	*b*
@2	*a*	*b*
@3	*a*	*b*
@4	*e*	*b*

S	B	C
@5	*b*	*c*
@6	*b*	*c*
@7	*b*	*g*

$R \sqcap S$	A	B	C
@15	*a*	*b*	*c*
@26	*a*	*b*	*c*
@37	*a*	*b*	*g*

$R \sqcap S$	A	B	C
@45	*e*	*b*	*c*
@26	*a*	*b*	*c*
@37	*a*	*b*	*g*

$R \sqcap S$	A	B	C
@15	*a*	*b*	*c*
@26	*a*	*b*	*c*
@47	*e*	*b*	*g*

Clearly, $\sqcap$ is not a correct generalization of $\bowtie$ for non-compatible bags (: $\alpha(R) \neq \alpha(S)$): $\lceil R \sqcap S \rceil \not\equiv \lceil R \rceil \bowtie \lceil S \rceil$.

What we need is an operator $\times\!\!\times$ that joins for each pair of matching tuples as many duplicates as possible, by creating maximal overlap *per matching pair.* The join of R and S from example 8 should yield:

$R{\times\!\!\times}S$	A	B	C
@15	a	b	c
@26	a	b	c
@17	a	b	g
@45	e	b	c
@47	e	b	g

If the empty set is ignored, then $\times\!\!\times$ is the *minimal join*: all intermediate numbers of duplicates are meaningless. After all, if 7 duplicates are joined with 5 matching duplicates, the result could never have 4 (or 6 or 8) duplicates, could it?

Let's try the following definition:

Definition 23 *By induction over the number of tuples:*
$R{\times\!\!\times}\emptyset := \emptyset$
$\emptyset{\times\!\!\times}R := \emptyset$
$(R_1 \uplus S_1){\times\!\!\times}(R_2 \uplus S_2) := (R_1{\times\!\!\times}R_2) \uplus (R_1{\times\!\!\times}S_2) \uplus (S_1{\times\!\!\times}R_2) \uplus (S_1 \sqcap S_2)$,
where $R_i \sqcap S_i = \emptyset$ *and* $\lceil S_i \rceil = \{s_i\}$.

A large number of duplicates results if all duplicates in R are joined with all duplicates in S. For instance, if R contains 2 duplicates that match with 3 duplicates of a tuple in S, then the *extravagant join* $R \otimes S$ would contains 6 duplicates of the joined tuples, cf. example 9 below. Whereas $R{\times\!\!\times}S$ contains 2 duplicates.

Definition 24 *Let* $X := \alpha(R) \cap \alpha(S)$, *then:*
$R \otimes S := \{t \mid \exists r {\in_i} R : r =_X t,\ \exists s {\in_i} S : s =_X t,\ t(\textsc{sk}) = f(r(\textsc{sk}), s(\textsc{sk}))\}\dagger$,
where f is an injective function.

Finally, joining each duplicate tuple in R with any matching S-tuple yields the *directed join*:

Definition 25 $R \vec{\bowtie} S \;:=\; \{t \mid \exists r \in_i R :\ r =_X t \text{ and } t =_Y S\}$.
where $X = \alpha^*(R)$ *and* $Y = \alpha(R) \cap \alpha(S)$.

Some of these definitions become a lot simpler when one turns to the procedural interpretation. For instance, the differences between the minimal $R \rtimes\!\ltimes S$, the extravagant $R \otimes S$ and the directed $R \vec{\bowtie} S$ are caused by the various methods to compute the answer relation.

The computation of $R \rtimes\!\ltimes S$ requires that R and S are ordered. Going from one block of duplicates to the next, it joins as many duplicates of a pair of matching tuples from R and S as possible. In contrast, $R \otimes S$ combines each individual R-duplicate with *all* matching S-duplicates, so it can be computed without preordering R and S. Finally, $R \vec{\bowtie} S$ presupposes an index on (a subset of) X for S, to make sure that one can reach all matching S-tuples through X. Every duplicate of a tuple in R is joined with an arbitrary duplicate of each matching tuple in S. This will be especially appropriate when X is a foreign key from R to S and therefore the primary key of S.

Example 9 *Minimal, directed and extravagant join.*

R	A	B
@1	a	b
@2	a	b
@3	a	b
@4	e	b

S	B	C
@5	b	c
@6	b	c
@7	b	g

$R \bowtie S$	A	B	C
@11	a	b	c
@12	a	b	g
@13	e	b	c
@14	e	b	g

$R \otimes S$	A	B	C
@15	a	b	c
@25	a	b	c
@35	a	b	c
@16	a	b	c
@26	a	b	c
@36	a	b	c
@17	a	b	g
@27	a	b	g
@37	a	b	g
@45	e	b	c
@46	e	b	c
@47	e	b	g

$R \rtimes\!\ltimes S$	A	B	C
@15	a	b	c
@26	a	b	c
@17	a	b	g
@45	e	b	c
@47	e	b	g

$R \vec{\bowtie} S$	A	B	C
@15	a	b	c
@25	a	b	c
@35	a	b	c
@17	a	b	g
@27	a	b	g
@37	a	b	g
@45	e	b	c
@47	e	b	g

This example illustrates the explosion of duplicates that results from $\otimes$. The answer relation as a set only contains 4 tuples, whereas $R \otimes S$ has $6 + 3 + 2 + 1 = 12$ duplicates.

All versions of baggy join reduce to standard by duplicate removal, except for the internal join and the (rejected) generalized intersection, which may be empty inappropriately:

Lemma 5 $\lceil R \rceil \bowtie \lceil S \rceil \;\equiv\; \lceil R \rtimes\!\ltimes S \rceil \;\equiv\; \lceil R \vec{\bowtie} S \rceil \;\equiv\; \lceil R \otimes S \rceil$
$\lceil R \rceil \bowtie \lceil S \rceil \;\not\equiv\; \lceil R \bowtie_i S \rceil$
$\lceil R \rceil \bowtie \lceil S \rceil \;\not\equiv\; \lceil R \sqcap S \rceil$

The join in SQL seems to correspond with $\otimes$:

```
SELECT R.A, R.B, S.C FROM R,S WHERE R.B = S.B;
```

The standard join requires explicit duplicate removal, that is, the addition of DISTINCT.

The directed join corresponds with the intersection as selection reading mentioned above. In SQL:

```
    SELECT * FROM R
    WHERE B IN (SELECT B FROM S);
```

Alternatively:

```
    SELECT * FROM R
    WHERE EXISTS (SELECT * FROM S WHERE R.B = S.B);
```

Note, however, that R dictates the scope of the result, so $R\vec{\bowtie}S$ is only used with $\alpha(S) \subseteq \alpha(R)$ or under a projection: $\Pi_X(R\vec{\bowtie}S)$ for $X \subseteq \alpha(R)$.

The commutative join $\times\!\!\times$ corresponds in SQL to AND (: in selection conditions), though -again- not for arbitrary pairs of bags. It is invoked in situations where both arguments have either 0 or the same number of duplicates. Moreover, it requires compatible arguments, so we only need $\sqcap$ instead of $\times\!\!\times$.

3.6 Difference

As an operator on sets, the difference of two relations is a subset of the first one.

Definition 26 $R \setminus S \;:=\; \{t \mid t \in R \textit{ and } t \notin S\}^*$.

Extended with a system key, one still has $R\setminus_i S \subseteq R$:

Definition 27 $R\setminus_i S \;:=\; \{t \mid t\in_i R \textit{ and } t \notin_i S\}$.

The difference operator is naturally directed, thus favouring the directed interpretation: the number of remaining duplicates is determined by R. Still, one may either remove *all* R-tuples that match S-tuples, or only as many as there are duplicates in S. Both definitions will reduce to standard if R is relational. Counting the number of S-duplicates yields a *numerical difference*:

Definition 28 $R - S \;:=\; R\dagger\setminus_i S\dagger$,
for relocations $R\dagger$ *and* $S\dagger$ *with maximal overlap:*
$\forall r\in_i R\dagger, s\in_i S\dagger : \; r = s \Rightarrow r\in_i S\dagger$ or $s\in_i R\dagger$.

Note that according to this definition it is quite possible that $t \in R - \{t\}$ is true; to remove all S-duplicates in R one should use the *directed difference*:

Definition 29 $R\vec{-}S \;:=\; \{t \mid t\in_i R \textit{ and } t \notin S\}$.

Not all generalizations reduce to the standard difference operator after duplicate removal:

Lemma 6 $\lceil R \rceil \setminus \lceil S \rceil \;\equiv\; \lceil (R\vec{-}S) \rceil$
$\lceil R \rceil \setminus \lceil S \rceil \;\not\equiv\; \lceil (R\setminus_i S) \rceil$
$\lceil R \rceil \setminus \lceil S \rceil \;\not\equiv\; \lceil (R - S) \rceil$

The cause of these inequivalences is that S may not have enough duplicates (or, for $\backslash_i$, duplicates on the wrong location) to remove all duplicates in R. The following example proves the inadequacy of the $R - S$. (Note that $R\backslash_i S = R$.)

Example 10 *Numerical and directed difference.*

R	A	B	C
@1	a	a	a
@2	a	a	a
@3	a	b	e
@4	a	b	e
@5	c	d	f
@6	c	d	f
@7	c	d	f
@8	p	g	h
@9	p	g	h
@10	q	g	h

S	A	B	C
@11	a	b	e
@12	a	b	e
@13	a	b	f
@14	c	d	f
@15	c	d	f
@16	p	g	h
@17	q	g	h

$R - S$	A	B	C
@31	a	a	a
@32	a	a	a
@37	c	d	f
@39	p	g	h

$R\overset{\rightarrow}{-}S$	A	B	C
@41	a	a	a
@42	a	a	a

The directed difference $R\overset{\rightarrow}{-}S$ corresponds with the difference construction that must be used in standard SQL:

```
SELECT * FROM R
WHERE R.A NOT IN
   (SELECT S.A FROM S WHERE R.A = S.A);
```

In this example both relations have a primary key A. The result will contain all duplicates in R that match no tuple in S. The standard operator requires the addition of the option `DISTINCT`. In the dialect Oracle, the latter can be achieved with the operator **`MINUS`**. The 'numerical' difference operation $R - S$ corresponds to `AND NOT` as employed in selection conditions, once more only for bags with 0 or the same number of duplicates. However, under those restrictions one can employ $R\overset{\rightarrow}{-}S$ just as well, so there is really no practical need for $-$.

3.7 Renaming, Calculate

The renaming operator has no effect whatsoever on SK, since SK is excluded from the set of renameable attributes. One would not expect the number of duplicates to be changed when some column is renamed, so the standard definition is what is needed.

Similarly, if new attributes are calculated out of given ones, then the result is not affected by the presence of duplicates, so the standard definition -trivially generalized- will do.

4 Additional operators

Apart from the set operator $\lceil - \rceil$ and the relocation operator $\dagger$, we need an operator to sort the tuples in a bag, where some ordering relation $\leq$ on X-tuples is presupposed.

Definition 30 $\text{SORT}_X(R) \;:=\; R\dagger$,
for a relocation of R *such that* $\forall r, t \in_i R\dagger : \; r(X) \leq t(X) \iff r(\text{SK}) \leq t(\text{SK})$.

Hence the result of the ordering operator is vacuous from a relational point of view, as it should be, but it is relevant for the implementation. Note that sorting makes sense for relations as well as for bags, only in the absence of a system key there is no way to express sorting at all!

Aggregate operators should be defined as relational or baggy operators, so as to preserve the clean ontology of the relational database model (: *everything is a relation*). This can be done by means of aggregate operators such as $\text{AVG}_{A:X}$, asking for the average A-value per X. This operator computes a relation with attributes X and $\text{AVG}(A)$ (cf. [9], pg 91 ff.).

Unfortunately, in SQL aggregates are expressed in a rather roundabout manner, namely by means of the construction `SELECT AVG(A) FROM R GROUP BY X`. For the sake of the `GROUP BY` option, we could use a grouping operator, that turns a relation into a set of equivalence classes. The result is no longer a relation or bag. The occurrence of this type of non-relations in SQL is one of the many ill-founded deviations from the relational model. The formal definition of this grouping operator would be:

$$\Upsilon_X(R) \;:=\; \{G_r \mid r \in R\}, \text{ where } G_r \;:=\; \{s \in_i R \mid s =_X r\}.$$

In order to remain as relational as possible, we will reformulate grouping by means of relations that are not in first normal form. Hence:

Definition 31 $\Upsilon_X(R) \;:=\; \{g : \alpha^*(R) \to \mathcal{D}^* \mid$
$\exists r \in R, \forall A \in X : \; g(A) = r(A) \;\&\; \forall A \notin X : \; g(A) = \ll r(A) \mid r =_X g \gg\}$

In this definition $\ll r(A) \mid r =_X g \gg$ is an ordered finite sequence, the order of which will be derived from the SK ordering of R. Obviously, the sequences $g(A)$ and $g(B)$, for attributes $A, B \notin X$, are of equal length, whereas for 2 tuples g_1 and g_2 the lengths of $g_1(A)$ and $g_2(A)$ may differ. The length of $g(A)$ equals the cardinality of G_r, the number of duplicates that are X-equal to r.

$\Upsilon_X(R)$ cannot be formulated by means of set-valued attributes, even if aggregates are restricted to atomic attributes (: to exclude complex terms like $\text{AVG}(\text{SALARY} + \text{BONUS})$, since there may be duplicates in each individual column of R. So we cannot employ Parendeans *nested relations* (cf. [11]). The choice between equivalence classes or sequence-valued attributes is pragmatic: we need to be able to define aggregates in a comprehensible manner.

Note that well-formedness conditions must ensure that sequenced-valued attributes are only compared/combined with attributes of the same format.

The problems caused by aggregate functions are not specific for duplicates. Still, the introduction of duplicates is often motivated by arguments on the proper treatment of aggregates. In the sequel we will ignore aggregates, since they are not part of the nucleus of a Codd-complete query language. Only when we get to the semantics of SQL, will they reappear into consideration.

identity	$\lceil R \rceil$	R	$R\dagger$		$\mathrm{SORT}_X(R)$
projection	$\Pi_X(R)$	$\Pi_{iX}(R)$	$\pi_X(R)$		
selection	$\sigma_\varphi(R)$				
calculate	$\kappa_\varphi(R)$				
renaming	$R[B/A]$				
union	$R \cup S$	$R \cup_i S$	$R \sqcup S$	$R \uplus S$	
intersection	$R \cap S$	$R \cap_i S$	$R \sqcap S$	$R \vec{\sqcap} S$	
join	$R \bowtie S$	$R \bowtie_i S$	$R \times\!\times S$	$R \vec{\bowtie} S$	$R \otimes S$
difference	$R \setminus S$	$R \setminus_i S$	$R - S$	$R \vec{-} S$	

Figure 1: Baggy operators

5 Overview in terms of duplicates

The set of algebraic operators has been enlarged to roughly 4 variant per standard operator (cf. figure 1). Naturally, one may not want to support all of them, but before a choice can be made, one must establish dependencies: which choices go together? It will be argued in the next section that one should choose alternatives that satisfy as many standard equivalences as possible.

In order to verify or falsify these equivalence we need a simplified interpretation that can guide our search. Suppose all base relations are singletons, which are implemented as *singleton bags*, a number of duplicates of a single tuple. We will interpret all terms by the number of duplicates in their *singleton* interpretation. The difference between alternatives will be reflected by the resulting number of duplicates. For instance, if R is a singleton bag, consisting of m duplicates of a single tuple, and S is a singleton bag of n duplicates, then $R \sqcup S$ contains $\mathtt{max}\{m, n\}$ tuples, whereas the number of tuples in $R \uplus S$ is $m + n$. For intersection the result will of course vary, depending on the equality of the duplicates in R and S, so the result is either 0 or a function of m and n. The results of this interpretation are listed in figure 2.

Strangely enough, the singleton interpretation of $R \otimes S$, which is $m * n$, is even larger than the largest result of the extended interpretation. This is due to the fact that matching tuples that differ on SK are combined, but there is no Cartesian product of extended relations: there is only one system key SK.

It may seem that $R \vec{\sqcap} S$ may have a singleton interpretation that is too large, as well, namely if $m < n$. However, this operator is derived from $\sigma_\varphi(R)$, so its singleton interpretation is one of $\{0, m\}$. The same thing holds for $R \vec{\bowtie} S$ and $R \vec{-} S$.

The singleton interpretation does not suffice for projection: increasing the number of duplicates presupposes that 2 tuples coincide on their projected part. Hence a double interpretation is added for projection.

The singleton interpretation is closely related to the concept of multisets -as opposed to sets- of tuples, that is, tuples extended with a finite multiplicity. Let us call the resulting annotated relations *multi-relations*.

Obviously, the extended operations (: $\cup_i, \ldots$) are undefined on multi-relations, but

type	standard	extended	minimal	maximal	directed
identity	$\lceil R \rceil$	R	$R\dagger$		$\texttt{SORT}_X(R)$
#	1	m	m		m
#	2	m_1, m_2			
identity	$\lceil S \rceil$	S			
#	1	n			
projection	$\Pi_X(R)$	$\Pi_{iX}(R)$	$\pi_X(R)$		
#	1	m	m		
	$\Pi_X(\{r_1, r_2\})$	$\Pi_{iX}(\{r_1, r_2\})$	$\pi_X(\{r_1, r_2\})$		
#	1	$m_1 + m_2$	$m_1 + m_2$		
#	2	m_1, m_2	m_1, m_2		
selection	$\sigma_\varphi(R)$	$\sigma_\varphi(R)$			
#	1	m			
#	0	0			
calculate	$\kappa_\varphi(R)$	$\kappa_\varphi(R)$			
#	1	m			
renaming	$R[B/A]$	$R[B/A]$			
#	1	m			
union	$R \cup S$	$R \cup_i S$	$R \sqcup S$	$R \uplus S$	
#	2	m, n	m, n	m, n	
#	1	$\texttt{max}\{m, n\}; m + n$	$\texttt{max}\{m, n\}$	$m + n$	
intersection	$R \cap S$	$R \cap_i S$		$R \sqcap S$	$R \vec{\cap} S$
#	1	$0; \texttt{min}\{m, n\}$		$\texttt{min}\{m, n\}$	m
#	0	0		0	0
join	$R \bowtie S$	$R \bowtie_i S$	$R \times\!\!\times S$	$R \otimes S$	$R \vec{\bowtie} S$
#	1	$0; \texttt{min}\{m, n\}$	$\texttt{min}\{m, n\}$	$m \times n$	m
#	0	0	0	0	0
difference	$R \setminus S$	$R \setminus_i S$	$R - S$		$R \vec{-} S$
#	1	m	m		m
#	0	$0 \; ; \; m$	$\texttt{max}\{m - n, 0\}$		0

Figure 2: The singleton interpretation

all remaining operations can be applied on multi-relations with the help of the singleton interpretation of bags.

For instance, the join of the multi-relations R and S is defined as usual over the relational attributes, whereas the multiplicity of the tuples in the result is a function of the participating multiplicities: 1, m, $min\{m,n\}$, $m \times n$, for set, directed, minimal and maximal join respectively. The main interest of all preceeding definitions is to clarify the reasons why this function must be one of these but not the average, sum or whatever. Similar arguments can be given for all other operations.

6 Relational equivalences

There exists a large set of standard equivalences that are used to establish a normal form for the sake of query optimization (see [5]). These equivalences need to be reconsidered in the context of duplicates. Some definitions may seem to make sense, but if they require the sacrifice of Boolean laws, or selection distribution, or whatever, then they cannot be offered as a reasonable query language.

In the sequel, we make use of the following implication:

If 2 terms R and S are equivalent, $R \equiv S$, then
1. their standard reducts are equivalent on database instances (: without SK)
2. they yield the same number of duplicates under the singleton interpretation.

Unfortunately, the converse implication is invalid, at least in the presence of the projection operator. It is not hard to see that $R \bowtie \Pi_Y(S) \;\equiv\; \Pi_{XY}(R \bowtie S)$, provided that Y includes all common attributes $\alpha(R) \cap \alpha(S)$ and $X = \alpha(R) \setminus Y$. This equivalences is true for all baggy join operators *only* if both bags are singleton bags (: duplicates of a single tuple). If S contains 2 tuples, then both $\times\!\!\times$ and $\vec{\bowtie}$ may fail to obey this property.

Suppose $s_1 =_Y s_2$ with n_1 and n_2 duplicates respectively. Then $\Pi_Y(S)$ has $n_1 + n_2$ duplicates, but $R\vec{\bowtie}\Pi_Y(S)$ only m, whereas $R\vec{\bowtie}S$ has 2 tuples with m duplicates each, yielding a total of $2m$ for $\Pi_{XY}(R\vec{\bowtie}S)$. Similarly for $\times\!\!\times$ when $n_i < m < n_1 + n_2$.

It is not hard to see that the behaviour of $\sqcap$ and $\times\!\!\times$ and of $\vec{\sqcap}$ and $\vec{\bowtie}$ with respect to duplicates is identical. Hence we will only consider one of each pair, in particular, $\sqcap$ as the dual of $\sqcup$ and $\vec{\bowtie}$ as it generalizes $\vec{\sqcap}$ for uncompatible terms. We will divert from this default choice in cases where scope considerations force our hand.

Let us start with the basic set equivalences. We will use the standard operator, if all versions on bags behave the same, or if there is only a single exception listed explicitly.

Lemma 7 *Commutativity & associativity.*

- $R \cup S \;\equiv\; S \cup R$
- $R \bowtie S \;\equiv\; S \bowtie R$
  - $R\vec{\bowtie}S \;\not\equiv\; S\vec{\bowtie}R$
- $(R \cup S) \cup T \;\equiv\; R \cup (S \cup T)$
- $(R \bowtie S) \bowtie T \;\equiv\; R \bowtie (S \bowtie T)$

The extravagant $\otimes$ operator, which yields $m * n$ duplicates, is the only join operator that distributes over $\uplus$. Moreover, there is no union operator that in turn distributes over $\otimes$. Distributivity is rather a basic property, which we would not like to lose.

Lemma 8 *Distributivity.*

- $R \bowtie (S \cup T) \;\equiv\; (R \bowtie S) \cup (R \bowtie T)$
  - $R \sqcap (S \uplus T) \;\not\equiv\; (R \sqcap S) \uplus (R \sqcap T)$
  - $R \overset{\to}{\sqcap} (S \uplus T) \;\not\equiv\; (R \overset{\to}{\sqcap} S) \uplus (R \overset{\to}{\sqcap} T)$
- $R \cup (S \bowtie T) \;\equiv\; (R \cup S) \bowtie (R \cup T)$
  - $R \sqcup (S \otimes T) \;\not\equiv\; (R \sqcup S) \otimes (R \sqcup T)$
  - $R \uplus (S \otimes T) \;\not\equiv\; (R \uplus S) \otimes (R \uplus T)$

Next we will study the universal relations and their complement, the empty set. Let U be a universal relation with adequate scope, that is, $\alpha(R) \cup \alpha(S) \cup \alpha(T) \subseteq \alpha(U)$ and $U := \mathcal{D}^{\alpha(U)}$. U is implemented on an arbitrary location, but no duplicates.

Lemma 9 *Idempotency, units.*

- $R \;\equiv\; R \sqcap R \;\equiv\; R \overset{\to}{\bowtie} R \;\not\equiv\; R \otimes R$
- $R \;\equiv\; R \sqcup R \;\not\equiv\; R \uplus R$
- $R \;\equiv\; R \otimes U \;\equiv\; R \overset{\to}{\bowtie} U \;\not\equiv\; R \sqcap U \;\equiv\; \lceil R \rceil$
- $U \;\not\equiv\; R \sqcup U \;\not\equiv\; R \uplus U \;\not\equiv\; U$
- $\emptyset \;\equiv\; R \sqcap \emptyset \;\equiv\; R \overset{\to}{\bowtie} \emptyset \;\equiv\; \emptyset \overset{\to}{\bowtie} R \;\equiv\; R \otimes \emptyset$
- $\emptyset \;\equiv\; R \overset{\to}{-} U \;\not\equiv\; R - U$
- $R \;\equiv\; R \sqcup \emptyset \;\equiv\; R \uplus \emptyset \;\equiv\; R - \emptyset \;\equiv\; R \overset{\to}{-} \emptyset$

Obviously, if $\lceil - \rceil$ is added as outermost operator, all turn into valid equivalences, the exception being $\lceil \emptyset \rceil \;\not\equiv\; \lceil R - U \rceil$. Indeed, as soon as the difference operator is considered, things deteriorate. The directed version behaves best:

Lemma 10 *Difference cascade.*

- $(R \setminus S) \setminus T \;\equiv\; R \setminus (S \cup T) \;\equiv\; (R \setminus S) \bowtie (R \setminus T)$
  - $(R - S) - T \;\not\equiv\; (R - S) \bowtie (R - T)$ *(for any join)*
  - $(R - S) - T \;\not\equiv\; R - (S \sqcup T)$
  - $(R \overset{\to}{-} S) \overset{\to}{-} T \;\not\equiv\; (R \overset{\to}{-} S) \otimes (R \overset{\to}{-} T)$
- $R \setminus (S \setminus T) \;\equiv\; (R \setminus S) \cup (R \bowtie T)$

- $R - (S - T) \;\not\equiv\; (R - S) \cup (R \cap T)$ *(for any join and union)*
- $R \overset{\rightarrow}{-} (S \overset{\rightarrow}{-} T) \;\not\equiv\; (R \overset{\rightarrow}{-} S) \uplus (R \cap T)$ *(for any join)*
- $R \overset{\rightarrow}{-} (S \overset{\rightarrow}{-} T) \;\not\equiv\; (R \overset{\rightarrow}{-} S) \cup (R \otimes T)$ *(for any union)*

The difference operator distributes over union and intersection in other combinations as well.

Lemma 11 *Difference distribution.*

- $(R \cup S) \setminus T \;\equiv\; (R \setminus T) \cup (S \setminus T)$
  - $(R \cup S) - T \;\not\equiv\; (R - T) \cup (S - T)$ *(for any union)*
- $R \setminus (S \bowtie T) \;\equiv\; (R \setminus S) \cup (R \setminus T)$
  - $R \overset{\rightarrow}{-} (S \bowtie T) \;\not\equiv\; (R \overset{\rightarrow}{-} S) \uplus (R \overset{\rightarrow}{-} T)$
  - $R - (S \overset{\rightarrow}{\bowtie} T) \;\not\equiv\; (R - S) \cup (R - T)$
  - $R - (S \otimes T) \;\not\equiv\; (R - S) \cup (R - T)$
- $(R \bowtie S) \setminus T \;\equiv\; (R \setminus T) \bowtie (S \setminus T)$
  - $(R \overset{\rightarrow}{\bowtie} S) - T \;\not\equiv\; (R - T) \overset{\rightarrow}{\bowtie} (S - T)$
  - $(R \otimes S) - T \;\not\equiv\; (R - T) \otimes (S - T)$
- $(R \setminus S) \bowtie (T \setminus U) \;\equiv\; (((R \bowtie T) \setminus S) \setminus U)$
  - $(R - S) \bowtie (T - U) \;\not\equiv\; (((R \bowtie T) - S) - U)$ *(for any join)*

Obviously, any version of difference should validate $R \setminus R \;\equiv\; \emptyset$. Unfortunately, this is not sufficient to guarantee that $(R \bowtie S) \bowtie (R \setminus S) \;\equiv\; \emptyset$. This standard equivalence has 6 baggy equivalences, half of which are invalid, depending on the difference operation only. On the other hand, the equivalence $(R \cap S) \cup (R \setminus S) \;\equiv\; R$ has no less than 12 baggy versions, with only 4 of them valid.

Lemma 12 *Embedded complements.*

- $(R \bowtie S) \bowtie (R \setminus S) \;\equiv\; \emptyset$
  - $(R \bowtie S) \bowtie (R - S) \;\not\equiv\; \emptyset$ *(for any join)*
- $(R \cap S) \cup (R \setminus S) \;\equiv\; R$
  - $(R \overset{\rightarrow}{\cap} S) \uplus (R - S) \;\not\equiv\; R$
  - $(R \sqcap S) \sqcup (R - S) \;\not\equiv\; R$
  - $(R \sqcap S) \cup (R \overset{\rightarrow}{-} S) \;\not\equiv\; R$ *(for any union)*
  - $(R \otimes S) \cup (R \overset{\rightarrow}{-} S) \;\not\equiv\; R$ *(for any union)*
  - $(R \otimes S) \cup (R - S) \;\not\equiv\; R$ *(for any union)*

The selection operator can be applied in several equivalent manners. To give but the most obvious:

Lemma 13 *Selection laws.*

- $\sigma_{\phi \vee \psi}(R) \;\equiv\; \sigma_\phi(R) \cup \sigma_\psi(R)$
  - $\sigma_{\phi \vee \psi}(R) \;\not\equiv\; \sigma_\phi(R) \uplus \sigma_\psi(R)$
- $\sigma_{\phi \wedge \psi}(R) \;\equiv\; \sigma_\phi(R) \bowtie \sigma_\psi(R)$
  - $\sigma_{\phi \wedge \psi}(R) \;\not\equiv\; \sigma_\phi(R) \otimes \sigma_\psi(R)$

Under some scope restrictions, projection can be distributed over union and join.

Lemma 14 *Projection distribution.*

- $\Pi_X(R \cup S) \;\equiv\; \Pi_X(R) \cup \Pi_X(S)$
  - $\pi_X(R \sqcup S) \;\not\equiv\; \pi_X(R) \sqcup \pi_X(S)$
- $\Pi_X(R \bowtie S) \;\equiv\; \Pi_X(R) \bowtie \Pi_X(S)$ *(with $\alpha(R) \cap \alpha(S) \subseteq X$)*
  - $\pi_X(R \sqcap S) \;\not\equiv\; \pi_X(R) \sqcap \pi_X(S)$
  - $\pi_X(R \vec{\bowtie} S) \;\not\equiv\; \pi_X(R) \vec{\bowtie} \pi_X(S)$

The inequivalences in this section will be used to establish dependencies between the different baggy operations.

7 Dependent definitions

Given the large set of possibilities, it would be nice if an optimal combination of choices could be found that would make all others versions superfluous. Obviously, this presupposes that all versions are independent, which they are not. For instance, directed join can be defined in terms of the extravagant join, by removal of duplicates in the bag on the right-hand side: $R \vec{\bowtie} S \;\equiv\; R \otimes \lceil S \rceil$. However, the generating set itself should at least be optimal: it should satisfy all standard equivalences except of course those that are irrelevant in the context of duplicates.

We will not list all combinations that preserve some equivalence; there are too many. From the list given it appears that $\vec{-}$ behaves better than $-$ and $\sqcup$ better than $\uplus$. However, the latter is optimal as far as implementation is concerned: just copy the two lists one after the other. Hence we will start with the preferred implementations.

7.1 Feasible operations

Consider the language $\uplus$, $\overset{\rightarrow}{\bowtie}$, $\overset{\rightarrow}{-}$.

There are some important equivalences that are lost, the most obvious ones being commutativity of conjunction and idempotency of disjunction:

$$\begin{array}{l} R\overset{\rightarrow}{\bowtie}S \not\equiv S\overset{\rightarrow}{\bowtie}R \\ R \not\equiv R \uplus R \end{array}$$

Conjunction distributes over disjunction, but not the other way around. Moreover, neither conjunction nor difference can be pushed over the negative side of a difference:

$$\begin{array}{lclcl} R \uplus (S\overset{\rightarrow}{\bowtie}T) & \equiv & (R \uplus S)\overset{\rightarrow}{\bowtie}(R \uplus T) \\ R\overset{\rightarrow}{\bowtie}(S \uplus T) & \not\equiv & (R\overset{\rightarrow}{\bowtie}S) \uplus (R\overset{\rightarrow}{\bowtie}T) & \not\equiv & (R\overset{\rightarrow}{\bowtie}(S \uplus T)) \uplus (R\overset{\rightarrow}{\bowtie}(S \uplus T)) \\ R\overset{\rightarrow}{-}(S\overset{\rightarrow}{\bowtie}T) & \not\equiv & (R\overset{\rightarrow}{-}S) \uplus (R\overset{\rightarrow}{-}T) \\ R\overset{\rightarrow}{-}(S\overset{\rightarrow}{-}T) & \not\equiv & (R\overset{\rightarrow}{-}S) \uplus (R\overset{\rightarrow}{\bowtie}T) \end{array}$$

These inequivalences show that one cannot predict the number of duplicates after query optimization. Especially when duplicates are preserved in order to compute (later on) aggregates one must be wary of term rewriting.

Still, only the *number* of duplicates is affected, not their presence or absence. This means that for any $\uplus$, $\overset{\rightarrow}{\bowtie}$, $\overset{\rightarrow}{-}$ expression f over the base relations $R_1 \ldots R_k$ with standard form $\lceil f \rceil$ (: in terms of $\cup$, $\bowtie$, $\setminus$) one can easily prove

$$\lceil f(R_1, \ldots, R_k) \rceil \;\equiv\; \lceil f \rceil (R_1, \ldots, R_k)$$

In other words, as long as f is used as an implementation of $\lceil f \rceil$, keeping duplicates to increase performance, but without giving them any intuitive content, since they may be removed at any convenient moment, then the result -as a relation- is correct.

7.2 Maximal operations

Let us now consider the opposite choices: $\sqcup$, $\sqcap$, $-$.

The first two are well-behaved dual operators, but the numerical version of difference poses a problem. There are hardly any equivalences left, and moreover, some of these inequivalences even affect the presence or absence of tuples.

$$(R - S) - T \;\not\equiv\; R - (S \sqcup T) \;\equiv\; (R - S) \sqcap (R - T)$$

Proof: Let R contain 5 duplicates and both S and T 3, then there are 2 duplicates in $R - S$ and none in $(R - S) - T$. On the other hand, there are 3 duplicates in $S \sqcup T$, so 2 in $R - (S \sqcup T)$; again, there are 2 duplicates in $R - S$ and in $R - T$, so, once more, there are 2 duplicates in $(R - S) \sqcap (R - T)$.

The equivalence can be proven in general from the numerical truth of $m - \mathtt{max}\{n, k\} = \mathtt{min}\{m - n, m - k\}$.

■

Other inconvenient inequivalences include:

$$\begin{aligned}(R \sqcap S) \sqcap (R - S) &\not\equiv \emptyset\\ (R \sqcap S) \sqcup (R - S) &\not\equiv R\\ (R - (S - T) &\not\equiv (R - S) \sqcup (R \sqcap T)\end{aligned}$$

Let us not go into greater detail; this combination would lead to the sacrifice of many crucial equivalences. The result of a query is not preserved under optimization rewriting, not even after duplicate removal:

$$\begin{aligned}\lceil (R - S) - T \rceil &\not\equiv \lceil R - (S \sqcup T) \rceil\\ \lceil (R \sqcap S) \sqcap (R - S) \rceil &\not\equiv \emptyset\\ \lceil (R - (S - T)) \rceil &\not\equiv \lceil (R - S) \sqcup (R \sqcap T) \rceil\end{aligned}$$

On account of these set inequivalences, $\sqcup$, $\sqcap$, $-$ is *not* a correct implementation of $\cup$, $\bowtie$, $\setminus$.

7.3 Mixed operations

Consider $\sqcup$, $\sqcap$, $\vec{-}$.

The previous combination was rejected on account of the ill-behaved $-$, so if that one is replaced by the directed $\vec{-}$ we may hope for a reasonable result. Indeed, most equivalences are regained, the main exception being:

$$(R \sqcap S) \sqcup (R \vec{-} S) \not\equiv R$$

This standard equivalence is lost, as we miss all R-duplicates that are not in S, even if the tuple itself is. So we may end with a reduced bag of duplicates.

Obviously, this is not the only equivalence that is lost, but in all cases the presence or absence of a tuple is not affected, so this combination would constitute a correct implementation.

7.4 Directed operations

If difference is directed, then so should the join be; consider the language $\sqcup$, $\vec{\bowtie}$, $\vec{-}$.

Even though $\sqcup$ and $\sqcap$ are natural duals, $\sqcup$ and $\vec{\bowtie}$ also behave as duals. Moreover, all other equivalences are preserved as well, with the obvious exception of commutativity of conjunction (: $R \vec{\bowtie} S \not\equiv S \vec{\bowtie} R$), in particular:

$$(R \vec{\bowtie} S) \sqcup (R \vec{-} S) \equiv R$$

This combination behaves better than $\uplus$, $\vec{\bowtie}$, $\vec{-}$, since union distribution is preserved, and than $\sqcup$, $\sqcap$, $\vec{-}$, since embedded complements behave properly.

Nevertheless, none of these 3 combinations is perfect. In particular, neither the minimal nor the directed join validate

$$\pi_X(R \bowtie S) \equiv \pi_X(R) \bowtie \pi_X(S),$$

where this should be be valid as soon as X includes all shared attributes. Moreover, $\sqcup$ violates a similar constraint, $\pi_X(R \sqcup S) \not\equiv \pi_X(R) \sqcup \pi_X(S)$, whereas idempotency fails for $\uplus$.

7.5 Choice of operations

There are 8 more combinations of operations, but we need not consider every one of them. Some operations can be dismissed as incorrect or undesirable, regardless of the other choices made.

Difference

Note that there are no baggy operations that would combine with $-$ to a correct interpretation. In $R - S$ some S-tuples can be found, namely if R had more duplicates than S. As a consequence, for any $\bowtie$:

$$\lceil (R \bowtie S) \bowtie (R - S) \rceil \;\not\equiv\; \emptyset$$

This would be sufficient to dismiss the maximal difference $-$ as a possible implementation, except that there are equivalences that require $-$ instead of $\overset{\rightarrow}{-}$:

$$\begin{aligned} (R \sqcap S) \uplus (R - S) \;&\equiv\; R \\ (R \sqcap S) \uplus (R \overset{\rightarrow}{-} S) \;&\not\equiv\; R \end{aligned}$$

Still, losing duplicates is not half as bad as retaining unwanted tuples, so $-$ is not a serious candidate after all.

Union

The maximal union $\uplus$ has great implementational appeal, but it violates the classical equivalence $R \equiv R \cup R$. As a consequence, one cannot distribute a disjunctive selection condition to a union:

$$\sigma_{\phi \vee \psi}(R) \;\not\equiv\; \sigma_{\phi}(R) \uplus \sigma_{\psi}(R)$$

Moreover, the only baggy join over which $\uplus$ distributes is $\otimes$; both for $\overset{\rightarrow}{\bowtie}$ and $\sqcap$:

$$R \bowtie (S \uplus T) \;\not\equiv\; (R \bowtie S) \uplus (R \bowtie T)$$

In the next section we will study the connection with linear logic, in the hope that linear logic may provide a justification for this inequivalence.

Still, considering the relative acceptance of classical and linear logic, one might prefer to employ $\sqcup$. The main failure of $\sqcup$ is that it does not allow of projection distribution:

$$\pi_X(R \sqcup S) \not\equiv \pi_X(R) \sqcup \pi_X(S)$$

Moreover, it suffers from major implementation difficulties: one has to preorder R and S before they can be combined to $R \sqcup S$ (or to $R \sqcap S$, for that matter). On the other hand, if duplicates are implemented by means of multi-relations (: relations with the optional attribute *multiplicity*), then a simple min/max function suffices to extend the relational operation to a multi-relational operation.

Join

The extravagant join operation $\otimes$ suffers from idempotency problem analoguous to $\uplus$: it violates $R \equiv R \bowtie R$ and therefore lacks a union over which it can distribute:

$$R \cup (S \otimes T) \ \not\equiv \ (R \cup S) \otimes (R \cup T)$$

Moreover, one cannot express nested difference in terms of $\otimes$:

$$R \vec{-} (S \vec{-} T) \ \not\equiv \ (R \vec{-} S) \cup (R \otimes T)$$

And, of course, a conjunctive selection condition cannot be replaced by this type of join:

$$\sigma_{\phi \wedge \psi}(R) \ \not\equiv \ \sigma_{\phi}(R) \otimes \sigma_{\psi}(R)$$

On the other hand, $\sqcap$ requires multi-relations instead of bags, if they are to be implemented efficiently.

Finally, $\vec{\bowtie}$ is not commutative, which goes against many basic intuitions. One needs a totally different type of logic to account for non-commutativity.

7.6 Conclusion to this section

We have shown that the baggy operators fail to validate classical relational equivalences. One could try to find alternatives, baggy operators that do validate the classical equivalences, but the result could very well be irrelevant to practice.

Consider the `UNION ALL` option $\uplus$. The invalidity of $\sigma_{\phi \vee \psi}(R) \ \not\equiv \ \sigma_{\phi}(R) \uplus \sigma_{\psi}(R)$ can be found in **SQL**:

```
SELECT A, B FROM R WHERE C < 3
UNION ALL
SELECT A, B FROM R WHERE C < 4;
```

This query is *not* optimized before computation; it is *not equivalent* to the simple query (given that $C < 3 \vee C < 4 \ \equiv \ C < 4$):

```
SELECT A, B FROM R WHERE C < 4;
```

As a consequence, $\uplus$ is the operator we need; classical equivalences are of limited practical importance.

Still, if we are not to loose all possibility of query optimization, we should search for alternative systems of equivalences. In the next section, we try *linear logic*, a logic that abandones structural tautologies, such as: $\phi \ \equiv \ \phi \vee \phi$, which is similar to the present $R \not\equiv R \uplus R$.

8 Translation into logic

The motivation of this section is based on 2 observations:

- Linear logic is concerned with the structural difference between sets versus multisets of formulas, yielding a refinement of logical connectives.
- Relational operations are restricted versions of classical connectives, designed to guarantee a finite interpretation.

For a good introduction to linear logic we refer to [11]; for the connection between relational operators and classical logic to [1], [6], [8], [10], or [12]. The latter can be briefly summarized by the following identities:

$$\begin{aligned} \mathcal{I}(\varphi \vee \psi) &= \mathcal{I}(\varphi) \cup \mathcal{I}(\psi) \\ R \cup S &= \{t \mid t \in R \ \vee \ t \in S\} \end{aligned}$$

In other words, union *is* disjunction, only restricted to terms of equal scope in order to avoid infinite relations. Hence there is a close connection between the classical tautologies and the relational equivalences.

As we have seen in the previous sections, the presence of duplicates leads to some violations of relational equivalences. We have tried to find optimal combinations, so as to loose a minimal number of these classical equivalences. In this section we work contrarywise. If baggy operators do not correspond with classical connectives, which type of connectives do they match? In particular, can we translate the baggy algebra into linear logic and back again?

The intuitive idea would be to generalize the identities mentioned above. Linear logic contains 2 versions of disjunction, $+$ (: "parallel or") and $\oplus$ (: "additive or") and 2 versions of conjunction, $\star$ (: "tensor") and & (: "additive and"). (We have changed some notation; the symbols $\sqcup$ and $\sqcap$ used in [11] for $\oplus$ and & respectively are reserved in this work for communicative operators.)

How should we equate linear connectives to baggy operators? There are several possibilities, but some can be dismissed directly. The directed operator $\overrightarrow{\bowtie}$ is not symmetric, unlike the linear connectives, so it can never be linked to any of them. On account of the valid linear equivalence:

$$A \star (B \oplus C) \ \equiv \ (A \star B) \oplus (A \star C)$$

we can eliminate some combinations: if $\oplus = \uplus$, then $\star = \otimes$. Unfortunately, in [11] linear logic is not characterised by means of equivalences, but by means of deduction rules. Hence we cannot apply the results of § 6 directly, but they must be adapted to the format of linear logic. (The alternative would be to derive the linear equivalences.)

The sequent calculus for linear logic contains a left and right introduction rule for each connective. In figure 3 they are formulated in terms of union and intersection. Note that the rules **R**+ and **L**$\star$ are vacuous, because we opted to write $R, S \subseteq T, U$ as $R \cap S \subseteq T \cup U$. Obviously, we must reinterpret $R \subseteq S$ as baggy inclusion: there exists for some relocation $R\dagger$ of R such that $R\dagger \subseteq_i S$.

The remaining rules must now be verified for baggy operators, according to the connections that one proposes. Let us try to find context-free operators first.

context-free rules for union introduction:

L+ $U \cap R \subseteq T,\ U' \cap S \subseteq T' \ \Rightarrow\ U \cap U' \cap (R \cup S) \subseteq T \cup T'$

R+ $R \subseteq S \cup T \cup U \ \Rightarrow\ R \subseteq (S \cup T) \cup U$

context-free rules for join introduction:

L⋆ $U \cap R \cap S \subseteq T \ \Rightarrow\ U \cap (R \cap S) \subseteq T$

R⋆ $U \subseteq R \cup T,\ U' \subseteq S \cup T' \ \Rightarrow\ U \cap U' \subseteq (R \cap S) \cup T \cup T'$

context-sensitive rules for union* introduction:

L⊕ $U \cap R \subseteq T,\ U \cap S \subseteq T \ \Rightarrow\ U \cap (R \cup^* S) \subseteq T$

R⊕ $R \subseteq S \cup U \ \Rightarrow\ R \subseteq (S \cup^* T) \cup U$

context-sensitive rules for join* introduction:

L& $U \cap R \subseteq T \ \Rightarrow\ U \cap (R \cap^* S) \subseteq T$

R& $U \subseteq R \cup T,\ U \subseteq S \cup T \ \Rightarrow\ U \subseteq (R \cap^* S) \cup T$

Figure 3: Linear rules

8.1 The relational operators

Just to be sure we should try the standard join and union, which remove all duplicates.

$$\begin{aligned} \star &= \cap \\ + &= \cup \end{aligned}$$

The relational operators satisfy all ***structural rules*** of weakening and contraction, which disqualifies them from being linear context-free. (They are linear in the trivial sense that classical logic is also linear.)

$$\begin{aligned} &R \subseteq S \ \Rightarrow\ R \cap U \subseteq S \cup T \\ &R \cap R \cap U \subseteq S \cup S \cup T \ \Rightarrow\ R \cap U \subseteq S \cup T \end{aligned}$$

Obviously, both are ‘non-linear’ independent of each other; combining one relational operator with a baggy dual is no option.

8.2 Minimal union and join

One possibility would be the following connection:

$$\begin{aligned} \star &= \sqcap \\ + &= \sqcup \end{aligned}$$

Then we need to prove **L+**:

$$U \sqcap R \sqsubseteq T,\ U' \sqcap S \sqsubseteq T' \ \Rightarrow\ U \sqcap U' \sqcap (R \sqcup S) \sqsubseteq T \sqcup T'$$

This can be proven from the truth of:

$$min\{m_1, x\} \leq n_1,\ min\{m_2, y\} \leq n_2 \ \Rightarrow\ min\{m_1, m_2, max\{x, y\}\} \leq max\{n_1, n_2\}$$

Moreover, we need to check **R**$\star$:

$$U \sqsubseteq R \sqcup T,\ U' \sqsubseteq S \sqcup T' \ \Rightarrow\ U \sqcap U' \sqsubseteq (R \sqcap S) \sqcup T \sqcup T'$$

Obviously, this requires that the following is proven:

$$m_1 \leq max\{x, n_1\},\ m_2 \leq max\{y, n_2\} \ \Rightarrow\ min\{m_1, m_2\} \leq max\{min\{x, y\}, n_1, n_2\}$$

The other context-free rules were trivial, so this connection seems promising.

One of the inequivalences that requires justification is $(R \sqcap S) \sqcup (R \setminus S) \not\equiv R$. If baggy operators correspond to linear operators, then $R \setminus S$ equals either $\star\neg$ or $\&\neg$, context-free and context-sensitive difference respectively. Since there is no general purpose complement in relational algebra, we will check the derived rules for difference. The context-free rules for difference look like this:

$$U \sqcap R \sqsubseteq S \sqcup T \ \Rightarrow\ U \sqcap (R \setminus S) \sqsubseteq T$$
$$U \sqsubseteq R \sqcup T,\ U' \sqcap S \sqsubseteq T' \ \Rightarrow\ U \sqcap U' \sqsubseteq (R \setminus S) \sqcup T \sqcup T'$$

There is no difference operation on bags that satisfies these rules:

Example 11 *Numeral difference:*
1. Take a single tuple with 3 duplicates in U, 8 in R, 5 in S and 2 in T. Then $U \sqcap R$ has 3 duplicates, less than the 5 in $S \sqcup T$. However, $R - S$ has 8 - 5 = 3 duplicates, so $U \sqcap (R - S)$ has 3, which is more than 2, the number of duplicates in T.
2. Take a single tuple with 6 duplicates in U and U', 7 in R, 5 in S, 1 in T and 5 in T'. Then $R \sqcup T$ has 7 duplicates, more than U, and $U' \sqcap S$ has 5 duplicates, no more than T'. However, $R - S$ has 7 - 5 = 2 duplicates, so $(R - S) \sqcup T \sqcup T'$ has $max\{2, 1, 5\} = 5$, which is less than 6, the number of duplicates in $U \sqcap U'$.

Example 12 *Directed difference:*
1. Valid; for all natural numbers such that $min\{m, x\} \leq max\{y, n\}$, either $y = 0$ and $min\{m, x\} \leq n$ or $y \neq 0$ and $min\{m, 0\} = 0 \leq max\{y, n\}$.
2. Take a single tuple with 6 duplicates in U and U', 7 in R, 1 in S, T and T'. Then $R \sqcup T$ has 7 duplicates, more than U, and $U' \sqcap S$ has 1 duplicates, no more than T'. However, $R \overset{\rightharpoonup}{-} S$ has 0 duplicates, so $(R - S) \sqcup T \sqcup T'$ has $max\{0, 1, 1\} = 1$, which is less than 6, the number of duplicates in $U \sqcap U'$.

The context-sensitive rules for difference fare no better:

$$U \sqcap R \sqsubseteq T \ \Rightarrow\ U \sqcap (R \setminus S) \sqsubseteq T$$
$$U \sqsubseteq R \sqcup T,\ U \sqcap S \sqsubseteq T \ \Rightarrow\ U \sqsubseteq (R \setminus S) \sqcup T$$

Neither $-$ nor $\overset{\rightharpoonup}{-}$ satisfies the second rule:

Example 13 *Context-sensitive difference:*
Take a single tuple with 6 duplicates in U, 7 in R, 4 in S, and 5 in T. Then $R \sqcup T$ has 7 duplicates, more than U, and $U \sqcap S$ has 4 duplicates, no more than T. However, $R \overset{\rightarrow}{-} S$ has 0 duplicates and $R - S$ has 3, so $(R \setminus S) \sqcup T$ has 5, which is less than 6, the number of duplicates in U.

In the absence of an alternative difference operator, there is no linear difference operator and linear logic cannot help to explain the lost equivalence.

In fact, this connection does not work at all: if $\sqcap$ is context-free, then it should not be context-sensitive. However, it satisfies not only $\mathbf{R}\star$ but $\mathbf{L}\&$ as well:

$$U \sqcap R \subseteq T \;\Rightarrow\; U \sqcap (R \sqcap S) \subseteq T$$

Similarly, one can check $\mathbf{R}\&$, $\mathbf{R}\oplus$ and $\mathbf{L}\oplus$. Moreover, both operators satisfy the structural rules that turn linear logic into classical logic. In other words, if $\sqcap$ and $\sqcup$ are linear operators, they are classical operators as well.

8.3 Maximal union and product join

Now we will try the opposite choice:

$$\begin{aligned} \star &= \otimes \\ + &= \uplus \end{aligned}$$

We need to prove $\mathbf{R}\star$:

$$U \subseteq R \uplus T,\; U' \subseteq S \uplus T' \;\Rightarrow\; U \otimes U' \subseteq (R \otimes S) \uplus T \uplus T'$$

A counterexample is simple to find.

Example 14 *Suppose U and U' have 5 duplicates, R and S both 1 and T, T' 4. Then $U \otimes U'$ has 25 duplicates, but $(R \otimes S) \uplus T \uplus T'$ has only $1 \times 1 + 4 + 4 = 9$ duplicates.*

Obviously, things will not improve if $\uplus$ is replaced by the even weaker $\sqcup$:

$$\begin{aligned} \star &= \otimes \\ + &= \sqcup \end{aligned}$$

$\mathbf{R}\star$ now looks like this:

$$U \subseteq R \sqcup T,\; U' \subseteq S \sqcup T' \;\Rightarrow\; U \otimes U' \subseteq (R \otimes S) \sqcup T \sqcup T'$$

A counterexample:

Example 15 *Suppose U and U' have 5 duplicates, R and S both 3 and T, T' 6. Then $U \otimes U'$ has 25 duplicates, but $(R \otimes S) \sqcup T \sqcup T'$ has only $max\{3 \times 3, 6, 6\} = 9$ duplicates.*

8.4 Maximal union and minimal join

For completeness sake we should try the only remaining option:

$$\begin{aligned} \star &= \sqcap \\ + &= \uplus \end{aligned}$$

We need to prove **R**$\star$ and **L**$+$:

$$\begin{aligned} U \subseteq R \uplus T,\ U' \subseteq S \uplus T' &\Rightarrow U \sqcap U' \subseteq (R \sqcap S) \uplus T \uplus T' \\ U \sqcap R \subseteq T,\ U' \sqcap S \subseteq T' &\Rightarrow U \sqcap U' \sqcap (R \uplus S) \subseteq T \uplus T' \end{aligned}$$

These rules turn out to be valid, oddly enough:

$$\begin{aligned} m_1 \leq x + n_1,\ m_2 \leq y + n_2 &\Rightarrow min\{m_1, m_2\} \leq min\{x, y\} + n_1 + n_2 \\ min\{m_1, x\} \leq n_1,\ min\{m_2, y\} \leq n_2 &\Rightarrow min\{m_1, m_2, x + y\} \leq n_1 + n_2 \end{aligned}$$

So the minimal join in combination with maximal union satisfies the context-free rules, as it did in combination with minimal union. However, as before, it satisfies (3 out of 4) context-sensitive rules as well, **L**&, **R**& and **R**$\oplus$, respectively:

$$\begin{aligned} U \sqcap R \subseteq T &\Rightarrow U \sqcap (R \sqcap S) \subseteq T \\ U \subseteq R \uplus T,\ U \subseteq S \uplus T &\Rightarrow U \subseteq (R \sqcap S) \uplus T \\ R \subseteq S \uplus U &\Rightarrow R \subseteq (S \uplus T) \uplus U \end{aligned}$$

Obviously, $\sqcap$ still satisfies the structural rules, which makes it classical as well as linear. $\uplus$ satisfies weakening, but not contraction: $R \subseteq R \uplus R$, but $R \uplus R \not\subseteq R$. This is in accordance with the possibility to distribute $\sqcap$ over $\uplus$, and the impossibility to distribute $\uplus$ over $\sqcap$.

As before, we cannot add difference to the system, either as a context-free or as a context-sensitive operator. Counter-examples, analogous to the ones given in § 8.2, can be easily found. Hence we cannot rely on linear logic to explain the lack of an embedded complement, or the impossibility to simplify a difference cascade.

8.5 Conclusion to this section

This is where our investigation halts. There are no baggy operators that satisfy the context-free rules of linear logic, without also satisfying most of the context-sensitive rules.

In particular, there is no linear join. We have dismissed $\vec{\bowtie}$ for being non-symmetric. Both the relational join and the minimal join satisfy the structural rules of weakening and contraction and they satisfy the context-sensitive rules as well as the context-free rules. Finally, $\otimes$ does not even satisfy the context-free rules.

Nevertheless, the baggy versions of union and join can be identified with linear disjunction and conjunction, yielding a more or less substructural logic. Still, it has little explanatory power, since there is no baggy difference operator that corresponds with restricted linear negation. So the baggy algebra is not reducible to linear logic.

9 SQL queries

It is one of the wonders of relational database theory that SQL can claim to be relational *and* make use of the `DISTINCT` option. If you need an option to remove duplicates, you are dealing with bags of tuples, not with relations.

There exists a standard translation from the core of SQL into the relational algebra (cf. [12] and others). This translation ignores duplicates; in this section we study the role of duplicates in SQL.

Some familiarity with SQL is presupposed; for all details on the syntax of SQL we refer to [12] or [3]. The following summary of the syntax and semantics of SQL is based on [7].

9.1 Syntax

The general format of an SQL query is as follows:

```
SELECT [DISTINCT] X FROM R WHERE φ
GROUP BY Y HAVING ψ ORDER BY Z;
```

Not all key phrases are obligatory, in fact, only the `SELECT - FROM` part is indispensable. Queries can be combined by the `UNION [ALL]` operator and by nesting. The format of a subqueries is restricted by several additional conditions, such as: X is a single object term. The parameters are of the following types (cf. figure 5):

X	OTS	a list of object terms
R	RCS	a list of relation- and correlation names
φ	FORM	a formula, the selection condition
Y	ATTRS	a list of attributes
ψ	FORM	a formula
Z	ATTRS / VARS	a list of either attributes or variables

There are several well-formedness restrictions on the general format, such as: $YZ \subseteq X$ and X should be a subset of the *scope* of R, the set of attributes in R.

The syntactic structure of a query is illustrated in figure 4. The main types in an SQL query are listed in figure 5.

Some examples will illustrate the basic ingredients of an SQL query. FOT are functional object terms, such as "`salary` + `bonus`" and "`price` × 1, 04". AGT are terms involving aggregates (: `average(salary)`, `modal(name)`) and OT are object terms in general, including the shorthand $*$ for the list of all relevant attributes. Not all combinations of functions, aggregates and $*$ are possible; the hierarchy reflects terms as they appear in different places in a query.

FORM contains selection conditions like "`salary` > `bonus`", "`price` = f 20,08" and Boolean combinations of these. Conditions may involve subqueries, as in the following query:

```
SELECT * FROM clients
WHERE EXISTS ( SELECT name FROM members
WHERE members.donation = clients.bonus );
```

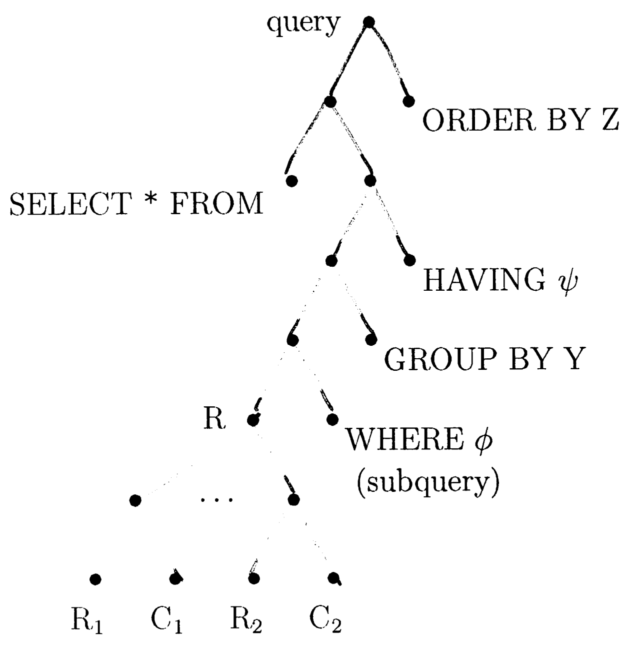

Figure 4: The structure of an SQL query

ATTR $\to$	[R.] A	R a relation name, A an attribute
AOT $\to$	ATTRS | CON	
FOT $\to$	AOT | f(FOTS)	f: $+, -, *, /$
AGT $\to$	`AGG` (`DISTINCT` AOT) |	`AGG`: `MIN, MAX, SUM, AVG`
	`AGG` ([`ALL`] FOT)	
OT $\to$	FOT | AGT | f(OTS) |	f: $+, -, *, /$
	$*$ | `COUNT(*)`	
RC $\to$	R | R C	R a relation, C a correlation name
FORM $\to$	$t_1 \theta t_2$ |	t_i FOT, $\theta : =, \neq, >, <, \geq, \leq$
	`EXISTS` Q |	Q a subquery
	`NOT` φ | φ `AND/OR` ψ | (φ) |	φ, ψ FORM
	$t_1\theta$ `ALL/ANY/SOME` Q |	
	t_1 `[NOT] BETWEEN` t_2 `AND` t_3 |	
	t_1 `[NOT] IN` $t*$ |	$t*$ FOTS (e.g.: t_1, t_2, t_3)
	t_1 `[NOT] IN` Q	
XXS $\to$	XX | XX, XXS	any type XX

Figure 5: The types in an SQL query

$\Vert R \Vert_r$	$:=$	$\Vert R\ R \Vert_r$
$\Vert R\ C \Vert_\emptyset$	$:=$	$\rho_C\mathcal{J}(R)$ (see below)
$\Vert S_1, \dots, S_n \Vert_r$	$:=$	$\Vert S_1 \Vert_r \otimes \dots \otimes \Vert S_n \Vert_r \otimes \{r\}$
$\Vert S\ \texttt{WHERE}\ \varphi \Vert_r$	$:=$	$\Vert S \Vert_r \vec{\bowtie} \Vert \varphi \Vert_r$ (see figure 7)
$\Vert S\ \texttt{GROUP BY}\ Y \Vert_r$	$:=$	$\Upsilon_X(\Vert S \Vert_r)$ (see pg 21)
$\Vert S\ \texttt{HAVING}\ \psi \Vert_r$	$:=$	$\{g \in_i \Vert S \Vert_r \mid g \bowtie r \models \psi\}$ (see below)
$\Vert \texttt{SELECT}\ X\ \texttt{FROM}\ S \Vert_r$	$:=$	$\pi_X(\Vert S \Vert_r)$ (X ATTRS)
$\Vert \texttt{SELECT DISTINCT}\ X\ \texttt{FROM}\ S \Vert_r$	$:=$	$\Pi_X(\Vert S \Vert_r)$ (X ATTRS)
$\Vert \texttt{SELECT}\ X\ \texttt{FROM}\ S \Vert_r$	$:=$	$\{s(X), s(\textsc{sk}) \mid s \in_i \Vert S \Vert_r\}$
$\Vert \texttt{SELECT DISTINCT}\ X\ \texttt{FROM}\ S \Vert_r$	$:=$	$\lceil \Vert \texttt{SELECT}\ X\ \texttt{FROM}\ S \Vert_r \rceil$
$\Vert \texttt{SELECT}\ *\ \texttt{FROM}\ S \Vert_r$	$:=$	$\Vert S \Vert_r$
$\Vert \texttt{SELECT}\ X\ \texttt{FROM}\ S \Vert_\emptyset$	$:=$	$\{g(X) \mid g \in \Vert S \Vert_\emptyset\}$ (X aggregated)
$\Vert S\ \texttt{ORDER BY}\ Z \Vert_\emptyset$	$:=$	$\textsc{sort}_X \Vert S \Vert_\emptyset$ (see pg 21)
$\Vert S\ \texttt{UNION ALL}\ T \Vert_\emptyset$	$:=$	$\Vert S \Vert_\emptyset \uplus \Vert S \Vert_\emptyset$
$\Vert S\ \texttt{UNION}\ T \Vert_\emptyset$	$:=$	$\Vert S \Vert_\emptyset \cup \Vert S \Vert_\emptyset$
$\Vert S\ \texttt{MINUS}\ T \Vert_\emptyset$	$:=$	$\Vert S \Vert_\emptyset \setminus \Vert S \Vert_\emptyset$

Figure 6: Baggy interpretation of SQL queries

Nested queries are an important source of confusion in SQL; it takes an expert to decide the equivalence of 2 nested queries.

9.2 Baggy interpretation

In this section, we will interpret SQL in the baggy model $< \mathcal{N}, \mathcal{D}, \mathcal{A}, \mathcal{J} >$. Recall that $\alpha^*(R)$ is the extended scope of R, the set of R's attributes plus SK. For each relation name R, $\mathcal{J}(R)$ is a set of extended tuples, an implementation of the set of tuples presently listed under R.

The interpretation of an SQL query will be a bag, as specified in figure 6. For the sake of subqueries, we add a tuple parameter to resolve attributes that are out of scope.

The (implicit) correlation name C ensures that all attributes are unique to their relation. In $\rho_C\mathcal{J}(R)$ all attributes are renamed from A to $C.A$, using R itself if C is not specified. Formally,

$$\begin{array}{l} \rho_C\mathcal{J}(R) \subseteq \mathcal{D}^{\{C.A \mid A \in \alpha(R)\} \cup \{\textsc{sk}\}} \text{ such that:} \\ \forall s \in \rho_C\mathcal{J}(R) \exists r \in_i \mathcal{J}(R) : s(\textsc{sk}) = r(\textsc{sk}) \ \&\ \forall A \in \alpha(R) : \ s(C.A) = r(A) \end{array}$$

We will identify $\mathcal{J}(R)$ with $\rho_R\mathcal{J}(R)$ whenever the context allows it, that is, we use A for $R.A$ as long as the prefix is unambiguous. This is common practice in SQL and, moreover, it is a syntactic problem unrelated to duplicates.

The interpretation of standard selection is straightforward; in SQL one can use a variety of selection-constructions, as listed in figure 7. Some notational details: the object that an extended tuple s assigns to a term t, $s(t)$, is defined by induction over t's structure: $s(A + B) := s(A) + s(B)$, and so on. $s(X)$ is short for $s(t_1), s(t_2), \dots$, if $X = t_1, t_2, \dots$. Scope restrictions on the language must assure that all attributes in t are bound by s.

$$\begin{array}{lll}
\| S\ \texttt{WHERE}\ t_1\theta t_2 \|_r & := & \sigma_{t_1\theta t_2}\| S \|_r \\
\| S\ \texttt{WHERE}\ t\ \texttt{IN}\ t* \|_r & := & \{s\in_i\| S \|_r \mid s(t) \in s(t*)\} \ \ \text{(similarly for } \texttt{BETWEEN}) \\
\| S\ \texttt{WHERE}\ \texttt{EXISTS}\ Q \|_r & := & \{s\in_i\| S \|_r \mid \| Q \|_{r\bowtie s} \neq \emptyset\} \\
\| S\ \texttt{WHERE}\ t\ \texttt{IN}\ Q \|_r & := & \{s\in_i\| S \|_r \mid s(t) \in \| Q \|_{r\bowtie s}\} \\
\| S\ \texttt{WHERE}\ t\ \theta\ \texttt{ANY/ALL}\ Q \|_r & := & \{s\in_i\| S \|_r \mid \exists/\forall q \in \| Q \|_{r\bowtie s} :\ s(t)\theta q(*)\} \\
\| S\ \texttt{WHERE}\ \texttt{NOT}\ \varphi \|_r & := & \| S \|_r \overset{\rightarrow}{-} \| S\ \texttt{WHERE}\ \varphi \|_r \\
\| S\ \texttt{WHERE}\ \varphi\ \texttt{AND}\ \psi \|_r & := & \| S\ \texttt{WHERE}\ \varphi \|_r \sqcap \| S\ \texttt{WHERE}\ \psi \|_r \\
\| S\ \texttt{WHERE}\ \varphi\ \texttt{OR}\ \psi \|_r & := & \| S\ \texttt{WHERE}\ \varphi \|_r \sqcup \| S\ \texttt{WHERE}\ \psi \|_r
\end{array}$$

Figure 7: Interpretation of $\| S\ \texttt{WHERE}\ \varphi \|_r$

The interpretation of HAVING is similar to WHERE, except that it allows aggregate terms. As a consequence, we must generalize the standard definition of $\sigma_{t_1\theta t_2}(R)$, adapting it to the presence of aggregated bags.

Suppose $R \subseteq (\mathcal{D}^*)^{\alpha(R)}$, so its tuples consist of finite sequences instead of single values (cf. § 4). A sequential generalization of functions and relations over these sequences must be defined in such a way that $r(A + B) = r(A) + r(B)$ for sequences and values alike. Hence, if $r(A)$ and $r(B)$ are both sequences:

$$\ll a_1, \ldots, a_n \gg \ + \ \ll b_1, \ldots, b_n \gg = \ll a_1 + b_1, \ldots, a_n + b_n \gg$$

Moreover, if $r(A)$ is a sequence and $r(B)$ is a single value:

$$\ll a_1, \ldots, a_n \gg \ + b \ = \ll a_1 + b, \ldots, a_n + b \gg$$

Similarly for other functions and relations; this generalization for the functions $+, -, \times, /$ and the relations θ $(: =, \leq, \ldots)$ is quite standard and unproblematic.

The aggregates pose no problem either: $r(\texttt{SUM}(A))$ is the sum of values in the sequence $r(A) \ = \ \ll a_1, \ldots, a_n \gg$, so $r(\texttt{SUM}(A)) := a_1 + \ldots + a_n$. By means of the DISTINCT option, one can remove duplicate values, that is, take the sum of the set of values in $r(A)$: $r(\texttt{SUM DISTINCT}(A)) := \Sigma\{a_1, \ldots, a_n\}$. Similarly for average, minimum and maximum. Moreover, $r(\texttt{COUNT}(*)) := n$.

Note that each r consists of sequences of fixed length n (depending on r), except for the GROUP BY attributes, that are single valued. Aggregate terms can be present in the HAVING clause as well as in the final SELECT. Obviously, there are a lot of syntactic details that should be clarified before this can be used as a proper interpretation (cf. [7]). The general idea of the present interpretation is illustrated in § 9.3 by means of examples.

9.3 Some SQL queries

The following examples (adapted from [7]) should give a fair idea how the present interpretation works.

Example 16 SELECT DISTINCT A FROM R;

$\|$ SELECT DISTINCT A FROM R $\|_\emptyset$

$= \Pi_A\| R \|_\emptyset$

$= \Pi_A\mathcal{J}(R)$

Example 17 SELECT A, B FROM R WHERE B = ‘b’;
$\|$ SELECT A, B FROM R WHERE B = ‘b’ $\|_\emptyset$
$= \pi_{AB}\|$ R WHERE B = ‘b’ $\|_\emptyset$
$= \pi_{AB}(\| R \|_\emptyset \vec{\bowtie} \| B = b \|_\emptyset)$
$= \pi_{AB}\{r \in_i \mathcal{J}(R) \mid r(B) = b\}$
$= \pi_{AB}\sigma_{B=b}\mathcal{J}(R)$

Example 18 SELECT A + 3 FROM R;
$\|$ SELECT A + 3 FROM R $\|_\emptyset$
$= \{r(\text{SK}), r(A+3) \mid r \in_i \| R \|_\emptyset\}$
$= \{r(\text{SK}), r(A) + 3 \mid r \in_i \mathcal{J}(R)\}$
$= \kappa_{A'=A+3}\pi_A\mathcal{J}(R)$

Example 19 SELECT S.B FROM R S, R T WHERE S.A = T.A AND T.B = ‘b’;
$\| Q \|_\emptyset$
$= \pi_{S.B}\|$ R S, R T WHERE S.A = T.A AND T.B = ‘b’ $\|_\emptyset$
$= \pi_{S.B}(\|$ R S, R T WHERE S.A = T.A $\|_\emptyset \sqcap \|$ R S, R T WHERE T.B = ‘b’ $\|_\emptyset)$
$= \pi_{S.B}(\|$ R S, R T $\|_\emptyset \vec{\bowtie} \|$ S.A = T.A $\|_\emptyset \sqcap \|$ R S, R T $\|_\emptyset \vec{\bowtie} \|$ T.B = ‘b’ $\|_\emptyset)$
$= \pi_{S.B}(\sigma_{S.A=T.A}(\|$ R S $\|_\emptyset \otimes \|$ R T $\|_\emptyset) \sqcap \sigma_{B=b}(\|$ R S $\|_\emptyset \otimes \|$ R T $\|_\emptyset))$
$= \pi_{S.B}(\sigma_{S.A=T.A \wedge B=b}(\rho_S\mathcal{J}(R) \otimes \rho_T\mathcal{J}(R)))$
$= \pi_B\sigma_{A=A' \wedge B'=b}(R \otimes R[A'/A, B'/B])$

Example 20 SELECT * FROM R WHERE A IN (SELECT A FROM S);
$\| Q \|_\emptyset$
$= \|$ R WHERE A IN (SELECT A FROM S) $\|_\emptyset$
$= \| R \|_\emptyset \vec{\bowtie} \|$ A IN (SELECT A FROM S) $\|_\emptyset$
$= \{r \in_i \| R \|_\emptyset \mid r(A) \in \|$ SELECT A FROM S $\|_r\}$
$= \{r \in_i \mathcal{J}(R) \mid r(A) \in \pi_A(\| S \|_r)\}$
$= \{r \in_i \mathcal{J}(R) \mid r(A) \in \pi_A(\mathcal{J}(S) \otimes \{r\})\}$
$= \{r \in_i \mathcal{J}(R) \mid r(A) \in \pi_A\mathcal{J}(S)\}$
$= \mathcal{J}(R) \vec{\bowtie} \pi_A\mathcal{J}(S)$

The following query is equivalent, as can be easily checked by comparing the interpretations. The renaming in example 21 is needed to avoid clashing variables, which are removed in example 20 by the projection on A (: the join-attribute).

Example 21 SELECT * FROM R WHERE EXISTS
(SELECT * FROM S WHERE S.A = R.A);
$\| Q \|_\emptyset$
$= \|$ R WHERE EXISTS (SELECT * FROM S WHERE S.A = R.A)) $\|_\emptyset$
$= \| R \|_\emptyset \vec{\bowtie} \|$ EXISTS (SELECT * FROM S WHERE S.A = R.A)) $\|_\emptyset$
$= \{r \in_i \| R \|_\emptyset \mid \|$ SELECT * FROM S WHERE S.A = R.A $\|_r \neq \emptyset\}$
$= \{r \in_i \| R \|_\emptyset \mid \|$ S WHERE S.A = R.A $\|_r \neq \emptyset\}$
$= \{r \in_i \| R \|_\emptyset \mid \sigma_{S.A=R.A}\| S \|_r \neq \emptyset\}$
$= \{r \in_i \rho_R\mathcal{J}(R) \mid \sigma_{S.A=R.A}(\rho_S\mathcal{J}(S) \otimes \{r\}) \neq \emptyset\}$
$= \{r \in_i \rho_R\mathcal{J}(R) \mid \{s \in_i \rho_S\mathcal{J}(S) \mid s(S.A) = r(R.A)\} \neq \emptyset\})$

$= \{r \in_i \rho_R \mathcal{J}(R) \mid \exists s \in_i \rho_S \mathcal{J}(S) : \; s(S.A) = r(R.A)\}$
$= \{r \in_i \rho_R \mathcal{J}(R) \mid r(R.A) \in \pi_{S.A}(\rho_S \mathcal{J}(S))\}$
$= (\mathcal{J}(R) \overset{\rightarrow}{\bowtie} \pi_A \mathcal{J}(S))[R.X/X]$

The following query is a reformulation of example 19. It is left to the reader to decide whether or not they are equivalent.

Example 22 `SELECT B FROM R WHERE A IN`
`(SELECT A FROM R WHERE B = 'b');`

$\| Q \|_\emptyset$
$= \pi_{S.B} \| \texttt{R WHERE A IN (SELECT A FROM R WHERE B = 'b')} \; \|_\emptyset$
$= \pi_B \{r \in_i \| R \|_\emptyset \mid r(A) \in \| \texttt{SELECT A FROM R WHERE B = 'b'} \|_r\}$
$=$
$= \pi_B \{r \in_i \mathcal{J}(R) \mid r(A) \in \pi_A(\sigma_{B=b} \mathcal{J}(R))\}$
$= \pi_B (R \overset{\rightarrow}{\bowtie} \pi_A \sigma_{B=b}(R))$

Example 23 `SELECT A, SUM(B) FROM R GROUP BY A;`

$\| Q \|_\emptyset$
$= \{g(A), g(\texttt{SUM}(B)) \mid g \in_i \| \texttt{R GROUP BY A} \|_\emptyset\}$
$= \{g(A), g(\texttt{SUM}(B)) \mid g \in_i \Upsilon_A \| R \|_\emptyset\}$
$= \{g(A), g(\texttt{SUM}(B)) \mid g \in_i \Upsilon_A \mathcal{J}(R)\}$
$= \{g(A), g(\texttt{SUM}(B)) \mid \exists r \in \mathcal{J}(R) : \; g(A) = r(A) \;\&\; g(B) = \ll t(B) \mid t(A) = g(A) \gg\}$
$= \{r(A), \Sigma(\ll t(B) \mid t(A) = r(A) \gg) \mid r \in \mathcal{J}(R)\}$
$= \texttt{SUM}_{B:A}(\mathcal{J}(R))$

The final step of this interpretation contains a (generalized) *relational aggregate* operator, computing the sum of B per A, by whatever algorithm performs best. As such it is but a convenient abbreviation of the previous step.

Aggregates in the `SELECT` clause require a `GROUP BY` declaration, to turn a relation into a set of sequence-tuples. Whenever there is no `GROUP BY` specified, the empty grouping is presupposed. The empty grouping turns a relation, a set of tuples, into a single sequence-tuple:

$\Upsilon_\emptyset \mathcal{J}(R) \; := \; \{<\ll t(A) \gg, \ldots, \ll t(B) \gg>\}$,
where for all $A \in \alpha(R)$: $\ll t(A) \gg := \pi_A \mathcal{J}(R)$ (: ordered according to SK.)

The following example employs this empty grouping construction.

Example 24 `SELECT AVG(A), SUM(B) FROM R;`

$\| Q \|_\emptyset$
$= \| \texttt{SELECT AVG(A), SUM(B) FROM R GROUP BY } \emptyset \; \|_\emptyset$
$= \{g(\texttt{AVG}(A)), g(\texttt{SUM}(B)) \mid g \in_i \| \texttt{R GROUP BY } \emptyset \; \|_\emptyset\}$
$= \{g(\texttt{AVG}(A)), g(\texttt{SUM}(B)) \mid g \in_i \Upsilon_\emptyset \mathcal{J}(R)\}$
$= \{g(\texttt{AVG}(A)), g(\texttt{SUM}(B)) \mid g \in_i \Upsilon_\emptyset \mathcal{J}(R) : \; g(A) = \ll t(A) \gg \;\&\; g(B) = \ll t(B) \gg\}$
$= \{\texttt{AVG}(\pi_A \mathcal{J}(R)), \texttt{SUM}(\pi_B \mathcal{J}(R))\}$
$= (\texttt{AVG}_A, \texttt{SUM}_B)(\mathcal{J}(R))$

Example 25 `SELECT A FROM R GROUP BY A HAVING COUNT(*) > 1;`

$\| Q \|_\emptyset$
$= \pi_A \| \texttt{R GROUP BY A HAVING COUNT(*)} > 1 \|_\emptyset$
$= \pi_A \{ g \in_i \| \texttt{R GROUP BY A} \|_\emptyset \mid g(\texttt{COUNT(*)}) > 1 \}$
$= \pi_A \{ g \in_i \Upsilon_A \mathcal{J}(R) \mid g(\texttt{COUNT(*)}) > 1 \}$
$= \{ g(A) \in_i \Upsilon_A \mathcal{J}(R) \mid g(\texttt{COUNT(*)}) > 1 \}$
$= \{ r(A) \in_i \mathcal{J}(R) \mid \#\{ s \in_i \mathcal{J}(R) : r(A) = s(A) \} > 1 \}$
$= \sigma_{\# > 1} \texttt{COUNT}_{*:A}(R)$

9.4 Baggy operators in SQL

Most of the baggy operators can be found in SQL, though some only in restricted circumstances. In § 3 the following correspondences where mentioned:

π_X	-	`SELECT X`
Π_X	-	`SELECT DISTINCT X`
σ	-	`WHERE`
$\cup$	-	`UNION`
$\uplus$	-	`UNION ALL`
$\sqcup$	-	`OR`
$\bowtie$	-	`DISTINCT * FROM ,`
$\otimes$	-	`FROM ,`
$\vec{\bowtie}$	-	`WHERE EXISTS`
$\sqcap$	-	`AND`
$\setminus$	-	`MINUS`
$\vec{-}$	-	`WHERE NOT EXISTS`

As an example, consider intersection. Let R and S be 2 tables over the same attributes. To abbreviate the query, we may assume that A is a primary key to both tables, but, to avoid triviality, this does not remove duplicates from the table. (An example might be a pair of views on AB defined by projection from a table with key C and constraints $C \rightarrow AB$, $A \rightarrow B$. The projection on AB can contain duplicates, though all tuples that agree on A agree on B as well.) If duplicates are removed automatically (: by definition, since projection yields a set of subtuples), then R and S would be relations and intersection can be expressed by several equivalent queries:

1. `SELECT * FROM R WHERE A IN (SELECT A FROM S);`

2. `SELECT * FROM S WHERE A IN (SELECT A FROM R);`

3. `SELECT R.A, R.B FROM R, S WHERE R.A = S.A;`

4. `SELECT DISTINCT R.A, R.B FROM R, S WHERE R.A = S.A;`

The equivalence of these queries depends of course on the key A. Smart query optimization might look for such equivalences to replace one by the other.

However, if R and S may contain duplicates, then these queries are no longer equivalent. Suppose R has 5 duplicates and S has 8 duplicates of a single tuple. Then there are

40 duplicates in $R \otimes S$, but only 5 in $R \overset{\rightarrow}{\bowtie} S$. As a consequence, the resulting sum of the A values changes by query optimalization! The results of the following queries on dBase IV reflect this phenomenon (for some interpretation with $r(A) = 1$):

1. `SELECT SUM(A) FROM R WHERE A IN (SELECT A FROM S)` $\Rightarrow 5$
2. `SELECT SUM(A) FROM S WHERE A IN (SELECT A FROM R)` $\Rightarrow 8$
3. `SELECT SUM(R.A) FROM R, S WHERE R.A = S.A` $\Rightarrow 40$
4. `SELECT SUM(DISTINCT(R.A)) FROM R, S WHERE R.A = S.A` $\Rightarrow 1$

These results may not surprise anyone familiar with SQL. Unfortunately, not every user of an SQL database will be familiar with the peculiarities of the language.

The following example must verify the present analysis of SQL. Take 3 unary relations or bags with n, m or 0 and k or 0 duplicates in their singleton interpretation respectively. In § 6 the following (in-)equivalences have been proven:

$$\begin{aligned}
R \overset{\rightarrow}{-} (S \overset{\rightarrow}{-} T) &\equiv (R \overset{\rightarrow}{-} S) \sqcup (R \overset{\rightarrow}{\bowtie} T) \\
R \overset{\rightarrow}{-} (S \overset{\rightarrow}{-} T) &\not\equiv (R \overset{\rightarrow}{-} S) \cup (R \overset{\rightarrow}{\bowtie} T) \\
R \overset{\rightarrow}{-} (S \overset{\rightarrow}{-} T) &\not\equiv (R \overset{\rightarrow}{-} S) \uplus (R \overset{\rightarrow}{\bowtie} T)
\end{aligned}$$

The number of duplicates in the result is given in the following table:

R	S	T	$R \overset{\rightarrow}{-} (S \overset{\rightarrow}{-} T)$	$(R \overset{\rightarrow}{-} S) \sqcup (R \overset{\rightarrow}{\bowtie} T)$	$(R \overset{\rightarrow}{-} S) \cup (R \overset{\rightarrow}{\bowtie} T)$	$(R \overset{\rightarrow}{-} S) \uplus (R \overset{\rightarrow}{\bowtie} T)$
n	0	0	n	n	1	n
n	m	0	0	0	0	0
n	0	p	n	n	1	$2n$
n	m	p	n	n	1	n

If the present translation of SQL into baggy algebra is correct, then this is reflected in SQL as follows:

1.
   ```
   CREATE VIEW V AS
   SELECT * FROM S WHERE A NOT IN (SELECT A FROM T);
   ```

2.
   ```
   SELECT * FROM R WHERE A NOT IN (SELECT A FROM V);
   ```

3.
   ```
   SELECT * FROM R WHERE ( A NOT IN (SELECT A FROM S)
   OR A IN (SELECT A FROM T WHERE) );
   ```

4.
   ```
   SELECT * FROM R WHERE A NOT IN (SELECT A FROM S)
   UNION SELECT * FROM R WHERE A IN (SELECT A FROM T);
   ```

5.
   ```
   SELECT * FROM R WHERE A NOT IN (SELECT A FROM S)
   UNION ALL SELECT * FROM R WHERE A IN (SELECT A FROM T);
   ```

Query (2) corresponds to $R \overset{\rightarrow}{-} (S \overset{\rightarrow}{-} T)$, (3) corresponds to $(R \overset{\rightarrow}{-} S) \sqcup (R \overset{\rightarrow}{\bowtie} S)$, (4) corresponds to $(R \overset{\rightarrow}{-} S) \cup (R \overset{\rightarrow}{\bowtie} S)$ and (5) corresponds to $(R \overset{\rightarrow}{-} S) \uplus (R \overset{\rightarrow}{\bowtie} S)$. According to the theory, one would expect the answers to resemble the pattern given above.

Note that it is not allowed in current implementations of SQL to define a view in terms of the last 2 queries. A possible explanation for this restriction would be that such views resist query optimization: rewriting the view definition will be dangerous in the context of duplicates.

9.5 Duplicates in SQL

The theoretical problems we encountered in the previous sections need not be relevant to the semantics of SQL. After all, the restrictions on the syntax or implementational constraints may be adequate in avoiding all problems. For example, consider the following inequivalence:

$$R \uplus (S \otimes T) \;\not\equiv\; (R \uplus S) \otimes (R \uplus T)$$

This poses no problem to SQL, as the righthand side cannot be translated into SQL, at least, not directly. However, there are views.

Let 3 tables R, S, T all have equal scope: A. All tables are projection views, which enables them to contain duplicates.

1.
   ```
   SELECT * FROM R UNION ALL
   SELECT S.A FROM S, T WHERE T.A = S.A;
   ```

2.
   ```
   CREATE VIEW Va AS
   SELECT * FROM R UNION ALL SELECT * FROM S;
   ```

3.
   ```
   CREATE VIEW Vb AS
   SELECT * FROM R UNION ALL SELECT * FROM T;
   ```

4.
   ```
   SELECT Va.A FROM Va, Vb WHERE Vb.A = Va.A;
   ```

The first query corresponds to the lefthand side, $R \uplus (S \otimes T)$, whereas the last query corresponds to the righthand side, $(R \uplus S) \otimes (R \uplus T)$. However, in current implementations this query and the 2 views `Va` and `Vb`, in terms of which it is defined, are refused. In the SQL handbook [10], there is no justification whatsoever for this restriction on views. The present analysis offers an explanation: the union view may contain duplicates, even if the underlying tables do not, so a query on such views resists query optimization.

The views being refused, it is impossible to check whether or not these queries yield identical results in practice. The question remains, should they be equivalent? According to the present interpretation the queries are only equivalent if duplicates are ignored, but the presence of `ALL` indicates that this presupposition is unwarranted. Still, if they are not equivalent, on what basis can one reformulate a query; how is query optimization to procede without equivalences? As we saw there is no straightforward alternative to classical logic to derive equivalences from.

Referring back to the examples 19 and 22 (see pg 41), their interpretations are not equivalent, since the first query yields rather more duplicates than the second:

$$\begin{aligned} &\pi_B \sigma_{A=A' \wedge B'=b}(R \otimes R[A'/A, B'/B]) \\ \not\equiv\; &\pi_B(R \overrightarrow{\bowtie} \pi_A \sigma_{B=b}(R)) \end{aligned}$$

As an example, consider the following instance:

R	A	B	C
@1	a	b	1
@2	a	b	2
@3	a	c	3
@4	a	c	4
@5	a	c	5

$\pi_B\sigma_{A=A'\wedge B'=b}(R \otimes R[A'/A, B'/B])$	B
(4)	b
(9)	c

$\pi_B(R\vec{\bowtie}\pi_A\sigma_{B=b}(R))$	B
(2)	b
(3)	c

Since the queries are both abstract versions of:

"Give the names of all employees that live in the same city as Mick"

there is hardly any point in listing duplicate names; none at all in squaring their number.

The following example is taken from [10]. It discusses optimization strategies that the user could try to improve response time, given that the present database systems are as yet unable to find the optimal equivalent query. The following 2 queries are equivalent and the latter improves the former (assuming the relevant indices):

```
SELECT * FROM R WHERE A = 1 OR B = 2;
SELECT * FROM R WHERE A = 1 UNION SELECT * FROM R WHERE B = 2;
```

The equivalence presupposes a key attribute in the selection, so for any non-unique R attribute C the following are *not* equivalent:

```
SELECT C FROM R WHERE A = 1 OR B = 2;
SELECT C FROM R WHERE A = 1 UNION SELECT * FROM R WHERE B = 2;
```

The former may contain duplicates that are removed by the `UNION` in the latter. The faster query turns out to be equivalent to:

```
SELECT DISTINCT C FROM R WHERE A = 1 OR B = 2;
```

In terms of baggy operators, one cannot replace $\pi_C\sigma_{\phi\vee\psi}(R)$ by $\pi_C\sigma_\phi(R) \cup \pi_C\sigma_\psi(R)$ (even though $\Pi_C\sigma_{\phi\vee\psi}(R)$ is equivalent to $\Pi_C\sigma_\phi(R) \cup \Pi_C\sigma_\psi(R)$), since the latter is equivalent to $\Pi_C\sigma_{\phi\vee\psi}(R)$. Moreover, $\pi_C\sigma_{\phi\vee\psi}(R)$ is not equivalent to $\pi_C\sigma_\phi(R) \uplus \pi_C\sigma_\psi(R)$ either, since $\pi_C\sigma_{\phi\vee\psi}(R)$ could be (: if C be a key attribute) equivalent to $\Pi_C\sigma_{\phi\vee\psi}(R)$, which can be replaced by $\pi_C\sigma_\phi(R) \uplus \pi_C\sigma_\psi(R)$ *only* if $\pi_C\sigma_{\phi\wedge\psi}(R)$ happens to yield $\emptyset$.

Seriously, what possible interest can the user have in duplicate rows in the result of the query that would justify these fineries? Surely, if the multiplicity of each C value is relevant, then it can be asked for explicitly, as in the SQL query:

```
SELECT C, COUNT(*) FROM R WHERE A = 1 OR B = 2 GROUP BY C;
```

The result of a query in a relational database should be a set, always. The presence of duplicates in the result of an SQL query is a concession to the implementation -duplicate removal impairs the response time of the query processor- for which the naive user is to pay a heavy price of confusion.

The following example is, again from [10], where it illustrates to counter-intuitive interplay between joins and aggregates. Suppose one needs to know for each A that is mentioned in table R the total of its B values as listed in table S. The obvious formulation would be:

```
SELECT R.A, SUM(B) FROM R, S WHERE R.A = S.A GROUP BY R.A;
```

The following -duplicate free- instance shows that this formulation is incorrect.

R	C	A
	c	a
	d	a
	e	a

S	A	B
	a	3
	a	4

$R \otimes S$	C	A	B
	c	a	3
	d	a	3
	e	a	3
	c	a	4
	d	a	4
	e	a	4

$\mathtt{SUM}_{B:A}(R \otimes S)$	A	SUM(B)
	a	21

Obviously, the desired sum is 7, but by the join construction each occurrence of A in R contributes to the sum. A correct reformulation of the query is not hard to find:

```
SELECT A, SUM(B) FROM S WHERE A IN (SELECT A FROM R) GROUP BY A;
```

This SQL query, however, reduces the role of R beyond what might be expected from the original NL formulation. A more user-friendly query employs a view:

```
CREATE VIEW V AS (SELECT DISTINCT A FROM R);
SELECT V.A, SUM(B) FROM V, S WHERE V.A = S.A GROUP BY V.A;
```

The equivalence of these 2 SQL queries is obvious from their interpretation in baggy terms:

$$\mathtt{SUM}_{B:A}(S \ltimes \pi_A(R)) \;\equiv\; \mathtt{SUM}_{B:A}(\Pi_A(R) \otimes S)$$

Note that the result indeed requires explicit duplicate removal in the view V, in order to yield the correct sum. The general strategy to improve response time by discouraging the use of the `DISTINCT` option is rather unfortunate in this context: it seduces the naive user to leave duplicates in views, even when this serves no meaningful purpose, leading to mistakes when such views are employed later on.

The present analysis of SQL in terms of baggy operators may not be much of an improvement from a practical point of view. Still, in trying to find possible inconsistencies, we encountered queries that are rejected by current implementations. We do not know whether or not the rejection of these queries has been motivated to the naive user at all. Obviously, simple examples can be offered to illustrate the problem and one can refer to the details of the implementation. The present analysis offers a *theoretical* motivation, based on term equivalences such as will be taught in any introductory course on database theory.

10 Conclusion

It is doubtful that the baggy database model is a conceptual improvement of the relational database model. Current implementations of the relational database model that fail to be fully relational suffer severe loss of comprehensibility exactly when dealing with the distinction between theory and practice; even if it takes additional procedural effort to avoid duplicates in the implemented relations, the effort has to be made for the sake of clarity.

Moreover, if one assumes that duplicates are essentially irrelevant, then all operators should be generalizations that reduce to standard by duplicate removal, that is, one could have used the standard operators instead of the baggy ones and derive the same result.

> If R_i are relations, and f is a baggy term with standard form $\lceil f \rceil$,
> then $\lceil f(R_1^*, \ldots, R_k^*) \rceil \ \equiv \ \lceil f \rceil (R_1, \ldots, R_k)$

This means that baggy operators can be used as harmless intermediates, implementations of the relational operators that are formally equivalent after a final duplicate removal. All baggy operations, except for $-$, meet this requirement.

Nevertheless, if duplicates are employed in order to compute aggregates (: average, maximum, etc.), then equivalent terms should contain equal numbers of duplicates, or else the equivalence cannot be used in query optimization. We have given examples of numerically invalid equivalences for a variety of baggy operations. The combination of the minimal union $\sqcup$ with directed join and difference (: $\vec{\bowtie}, \vec{\sqcap}, \vec{-}$) behaves best.

On the other hand, the minimal union is not very realistic from a practical point of view. A reasonable implementation would prefer $\uplus$, despite its violation of classical equivalences.

We have shown that baggy operators (: generalizations of relational operators) cannot be linked to generalizations of classical connectives, those of linear logic. As a consequence, we cannot explain the behaviour of baggy operators in terms of linear logic.

We applied the baggy database model to SQL, giving an interpretation of SQL queries in terms of bags. All baggy operators turn out to be useful, though some are restricted in their application by additional conditions on the scope of their arguments. For instance, $\sqcup$ and $\sqcap$ turn out to be applicable only if both arguments are selections of a common bag.

As a consequence, none of the inequivalences which disturbed us in the previous sections are problematic to SQL: one or both of the inequivalent terms cannot be expressed as an SQL query. By reference to the relevant inequivalence these otherwise unaccountable restrictions to the syntax are explained on a theoretical (rather than implementational) level, thus leading to a firmer understanding of the ideosyncrasies of SQL.

References

[1] E.F. Codd -'Relational Completeness of Database Sublanguages', in: *Database Systems*, R.Ruskin (ed), Prentice Hall Englewood (1972)

[2] E.F. Codd -*The Relational Database Model, version 2*, Addison-Wesley (1990)

[3] C.J. Date -*Relational Database: Selected Writings*, Addison-Wesley (1986)

[4] U. Dayal & P.A. Bernstein. -'On the Correct Translation of Update Operations on Relational Views', in: ACM Transactions of Database Systems, vol. 8 (3), pg 381-416 (1982)

[5] S.J. van Denneheuvel, G.R. Renardel de Lavalette, Z. Huang & K.L. Kwast -'A normal form for PCSJ expressions', in: Journal of the Zhengjiang Shipbuilding University, China (1991) Also as: ILLC CT-90-02 (1990)

[6] J.L. Hursch & C.J. Hursch -*Working with ORACLE*, TAB Books Inc. (1987)

[7] T. Imielinski & W. Lipski -'The relational model of data and cylindric algebras', in: Journal of Computer and System Sciences, vol. 28 (1984)

[8] P.G.M. Jansen -'De semantiek van SQL-queries', doctoraalscriptie, Universiteit van Amsterdam (1992)

[9] K.L. Kwast -*Unknown values in the relational database model*, Thesis, University of Amsterdam (1992)

[10] R.F. van der Lans. *het SQL leerboek*, Academic Service, Schoonhoven (1992)

[11] J. Parendaens, P. de Bra, M. Gyssens & D. van Gucht -*The structure of the Relational Database Model*, Springer-Verlag (1989)

[12] A.S. Troelstra -*Lectures on Linear Logic*, CSLI Lecture Notes 29 (1992)

[13] J.D. Ullman -*Principles of Database and Knowledge-Base Systems*, vol. 1 & 2, Computer Science Press (1989)

The ILLC Prepublication Series

1990

Logic, Semantics and Philosophy of Language

LP-90-01 Jaap van der Does	A Generalized Quantifier Logic for Naked Infinitives
LP-90-02 Jeroen Groenendijk, Martin Stokhof	Dynamic Montague Grammar
LP-90-03 Renate Bartsch	Concept Formation and Concept Composition
LP-90-04 Aarne Ranta	Intuitionistic Categorial Grammar
LP-90-05 Patrick Blackburn	Nominal Tense Logic
LP-90-06 Gennaro Chierchia	The Variablity of Impersonal Subjects
LP-90-07 Gennaro Chierchia	Anaphora and Dynamic Logic
LP-90-08 Herman Hendriks	Flexible Montague Grammar
LP-90-09 Paul Dekker	The Scope of Negation in Discourse, towards a Flexible Dynamic Montague grammar
LP-90-10 Theo M.V. Janssen	Models for Discourse Markers
LP-90-11 Johan van Benthem	General Dynamics
LP-90-12 Serge Lapierre	A Functional Partial Semantics for Intensional Logic
LP-90-13 Zhisheng Huang	Logics for Belief Dependence
LP-90-14 Jeroen Groenendijk, Martin Stokhof	Two Theories of Dynamic Semantics
LP-90-15 Maarten de Rijke	The Modal Logic of Inequality
LP-90-16 Zhisheng Huang, Karen Kwast	Awareness, Negation and Logical Omniscience
LP-90-17 Paul Dekker	Existential Disclosure, Implicit Arguments in Dynamic Semantics

Mathematical Logic and Foundations

ML-90-01 Harold Schellinx	Isomorphisms and Non-Isomorphisms of Graph Models
ML-90-02 Jaap van Oosten	A Semantical Proof of De Jongh's Theorem
ML-90-03 Yde Venema	Relational Games
ML-90-04 Maarten de Rijke	Unary Interpretability Logic
ML-90-05 Domenico Zambella	Sequences with Simple Initial Segments
ML-90-06 Jaap van Oosten	Extension of Lifschitz' Realizability to Higher Order Arithmetic, and a Solution to a Problem of F. Richman
ML-90-07 Maarten de Rijke	A Note on the Interpretability Logic of Finitely Axiomatized Theories
ML-90-08 Harold Schellinx	Some Syntactical Observations on Linear Logic
ML-90-09 Dick de Jongh, Duccio Pianigiani	Solution of a Problem of David Guaspari
ML-90-10 Michiel van Lambalgen	Randomness in Set Theory
ML-90-11 Paul C. Gilmore	The Consistency of an Extended NaDSet

Computation and Complexity Theory

CT-90-01 John Tromp, Peter van Emde Boas	Associative Storage Modification Machines
CT-90-02 Sieger van Denneheuvel, Gerard R. Renardel de Lavalette	A Normal Form for PCSJ Expressions
CT-90-03 Ricard Gavaldà, Leen Torenvliet, Osamu Watanabe, José L. Balcázar	Generalized Kolmogorov Complexity in Relativized Separations
CT-90-04 Harry Buhrman, Edith Spaan, Leen Torenvliet	Bounded Reductions
CT-90-05 Sieger van Denneheuvel, Karen Kwast	Efficient Normalization of Database and Constraint Expressions
CT-90-06 Michiel Smid, Peter van Emde Boas	Dynamic Data Structures on Multiple Storage Media, a Tutorial
CT-90-07 Kees Doets	Greatest Fixed Points of Logic Programs
CT-90-08 Fred de Geus, Ernest Rotterdam, Sieger van Denneheuvel, Peter van Emde Boas	Physiological Modelling using RL
CT-90-09 Roel de Vrijer	Unique Normal Forms for Combinatory Logic with Parallel Conditional, a case study in conditional rewriting

Other Prepublications

X-90-01 A.S. Troelstra	Remarks on Intuitionism and the Philosophy of Mathematics, Revised Version
X-90-02 Maarten de Rijke	Some Chapters on Interpretability Logic
X-90-03 L.D. Beklemishev	On the Complexity of Arithmetical Interpretations of Modal Formulae
X-90-04	Annual Report 1989
X-90-05 Valentin Shehtman	Derived Sets in Euclidean Spaces and Modal Logic
X-90-06 Valentin Goranko, Solomon Passy	Using the Universal Modality: Gains and Questions
X-90-07 V.Yu. Shavrukov	The Lindenbaum Fixed Point Algebra is Undecidable
X-90-08 L.D. Beklemishev	Provability Logics for Natural Turing Progressions of Arithmetical Theories
X-90-09 V.Yu. Shavrukov	On Rosser's Provability Predicate
X-90-10 Sieger van Denneheuvel, Peter van Emde Boas	An Overview of the Rule Language RL/1
X-90-11 Alessandra Carbone	Provable Fixed points in $I\Delta_0+\Omega_1$, revised version
X-90-12 Maarten de Rijke	Bi-Unary Interpretability Logic
X-90-13 K.N. Ignatiev	Dzhaparidze's Polymodal Logic: Arithmetical Completeness, Fixed Point Property, Craig's Property
X-90-14 L.A. Chagrova	Undecidable Problems in Correspondence Theory
X-90-15 A.S. Troelstra	Lectures on Linear Logic

1991

Logic, Semantics and Philosophy of Langauge

LP-91-01 Wiebe van der Hoek, Maarten de Rijke	Generalized Quantifiers and Modal Logic
LP-91-02 Frank Veltman	Defaults in Update Semantics
LP-91-03 Willem Groeneveld	Dynamic Semantics and Circular Propositions
LP-91-04 Makoto Kanazawa	The Lambek Calculus enriched with Additional Connectives
LP-91-05 Zhisheng Huang, Peter van Emde Boas	The Schoenmakers Paradox: Its Solution in a Belief Dependence Framework
LP-91-06 Zhisheng Huang, Peter van Emde Boas	Belief Dependence, Revision and Persistence
LP-91-07 Henk Verkuyl, Jaap van der Does	The Semantics of Plural Noun Phrases
LP-91-08 Víctor Sánchez Valencia	Categorial Grammar and Natural Reasoning
LP-91-09 Arthur Nieuwendijk	Semantics and Comparative Logic
LP-91-10 Johan van Benthem	Logic and the Flow of Information

Mathematical Logic and Foundations

ML-91-01 Yde Venema	Cylindric Modal Logic
ML-91-02 Alessandro Berarducci, Rineke Verbrugge	On the Metamathematics of Weak Theories
ML-91-03 Domenico Zambella	On the Proofs of Arithmetical Completeness for Interpretability Logic
ML-91-04 Raymond Hoofman, Harold Schellinx	Collapsing Graph Models by Preorders
ML-91-05 A.S. Troelstra	History of Constructivism in the Twentieth Century
ML-91-06 Inge Bethke	Finite Type Structures within Combinatory Algebras
ML-91-07 Yde Venema	Modal Derivation Rules
ML-91-08 Inge Bethke	Going Stable in Graph Models
ML-91-09 V.Yu. Shavrukov	A Note on the Diagonalizable Algebras of PA and ZF
ML-91-10 Maarten de Rijke, Yde Venema	Sahlqvist's Theorem for Boolean Algebras with Operators
ML-91-11 Rineke Verbrugge	Feasible Interpretability
ML-91-12 Johan van Benthem	Modal Frame Classes, revisited

Computation and Complexity Theory

CT-91-01 Ming Li, Paul M.B. Vitányi	Kolmogorov Complexity Arguments in Combinatorics
CT-91-02 Ming Li, John Tromp, Paul M.B. Vitányi	How to Share Concurrent Wait-Free Variables
CT-91-03 Ming Li, Paul M.B. Vitányi	Average Case Complexity under the Universal Distribution Equals Worst Case Complexity
CT-91-04 Sieger van Denneheuvel, Karen Kwast	Weak Equivalence
CT-91-05 Sieger van Denneheuvel, Karen Kwast	Weak Equivalence for Constraint Sets
CT-91-06 Edith Spaan	Census Techniques on Relativized Space Classes
CT-91-07 Karen L. Kwast	The Incomplete Database
CT-91-08 Kees Doets	Levationis Laus
CT-91-09 Ming Li, Paul M.B. Vitányi	Combinatorial Properties of Finite Sequences with high Kolmogorov Complexity
CT-91-10 John Tromp, Paul Vitányi	A Randomized Algorithm for Two-Process Wait-Free Test-and-Set
CT-91-11 Lane A. Hemachandra, Edith Spaan	Quasi-Injective Reductions
CT-91-12 Krzysztof R. Apt, Dino Pedreschi	Reasoning about Termination of Prolog Programs