Distributed and Parallel Databases, 4, 81–99 (1996)
© 1996 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Extending a Multi-Set Relational Algebra to a Parallel Environment

PAUL W.P.J. GREFEN grefen@cs.utwente.nl
JAN FLOKSTRA flokstra@cs.utwente.nl
University of Twente, Department of Computer Science, P.O. Box 217, 7500 AE Enschede, The Netherlands

Received September 30, 1994; Accepted May 18, 1995

Recommended by: Patrick Valduriez

Abstract. Parallel database systems will very probably be the future for high-performance data-intensive applications. In the past decade, many parallel database systems have been developed, together with many languages and approaches to specify operations in these systems. A common background is still missing, however. This paper proposes an extended relational algebra for this purpose, based on the well-known standard relational algebra. The extended algebra provides both complete database manipulation language features, and data distribution and process allocation primitives to describe parallelism. It is defined in terms of multi-sets of tuples to allow handling of duplicates and to obtain a close connection to the world of high-performance data processing. Due to its algebraic nature, the language is well suited for optimization and parallelization through expression rewriting. The proposed language can be used as a database manipulation language on its own, as has been done in the PRISMA parallel database project, or as a formal basis for other languages, like SQL.

Keywords: relational algebra, multi-set, query optimization, parallel query execution

1. Introduction

Parallel database systems will very probably be the future for high-performance data-intensive applications, as observed by many specialists in the field, e.g. [9]. In the past decade, many designs and implementations of parallel database systems have been realized. To describe operations to be performed by these systems, some kind of database manipulation language is necessary in which parallel operations can be specified. Many different languages have been proposed and used for this purpose, but a common background is still hard to find. Therefore, approaches are hard to compare, semantics of operations are often unclear, and languages and techniques are often strongly system-related.

This paper proposes an extended relational algebra for use with parallel database systems. The language has a close connection to the standard relational algebra as originally defined by Codd [6] and thus has a well-known and widely accepted background. The language is formally defined and has clear semantics. Further, the language has complete DML expressiveness, so it can either be used as a full-blown database manipulation language on its own, or as a formal background for other languages.

The language described in this paper is based on multi-sets of tuples, as opposed to the standard relational algebra that uses sets of tuples. The multi-set approach has three main advantages. In the first place, duplicate tuple semantics can be handled. This is important for supporting applications that model situations in which duplicate entities can exist. A set-based approach requires in these situations the addition of a synthetic attribute to

distinguish between duplicate entities, which can be considered unnatural and complicating. In the second place, the multi-set approach avoids duplicate elimination on intermediate results during query processing, which is necessary in a strictly set-based environment (e.g. after a relational projection operation). As duplicate removal is a costly operation, it should be avoided in situations where high performance is of premier importance, as is the case with parallel database systems. Also, the multi-set model allows for certain query optimizations that are not correct in a set-based environment. In the third place, multi-set semantics allow for the effective and efficient handling of aggregate functions. In set-based environments, aggregate functions can cause problems as these functions can result duplicate values.

Note that the multi-set approach described in this paper can be used for set-based applications too. In query operations, an operator for duplicate removal can be explicitly used (the *unique* operator introduced in Section 2.3). Base data can be guarded against duplicate tuples using integrity constraints (see e.g. [14]). A set-based model can thus be seen as a special case of the multi-set model presented in this paper.

1.1. Related work

Related research on a language for parallel database systems has been performed in the context of the Bubba project [3]. This project uses a sequential language called FAD that is mapped to an extended parallel language PFAD [16]. PFAD is not relational algebra based, and is aimed specifically towards a shared nothing message-passing architecture, including data send and receive constructs. In the HC16 project [4], relational algebra operations are mapped to low-level primitives for execution on a parallel machine. Here, the focus is on this mapping and the execution of the primitives, not on the formal definition of the algebra.

Research has also been performed on multi-set semantics for the relational model. Many of these proposals tend to be of a rather practical nature, however, lacking a formal mathematical background. The standard SQL definition uses multi-sets of tuples for example, but the semantics are defined in an operational way, lacking formal background and mathematical precision. Other approaches exist with a rather formal character, e.g. the one presented in [1], but these lack the direct connection to database practice. Further, approaches exist that try to capture multi-set semantics within a set-based relational theory. An example is the work in [17], where multi-sets are represented in subsets of columns of set-based relations.

In several places, the use of use of duplicates is argued against. Often, the argumentation is based on semantical problems with duplicate handling (as found in many SQL-based approaches). As the proposal in this paper gives clear and formally defined semantics, this argument doesn't hold here. In other places, a more fundamental position is presented. An example is The Third Manifesto [7], in which the use of duplicates is "outlawed" without argumentation other than the explicit opinion of the authors. We believe, however, that a well-formalized and complete multi-set relational model is a valid approach since it is a generalization of the set-based model with clear advantages as discussed above.

1.2. Organization of this paper

This paper is organized as follows. Section 2 describes the extended relational algebra in a sequential (centralized) database environment, giving the background for the main part of the paper. This section is based on the work presented in [15]. Section 3 extends the

sequential language to a parallel language by adding concepts for data distribution and process allocation. It is shown that these concepts nicely fit into the formal character of the language. Section 4 discusses expression rewriting in the extended algebra, as performed in query optimization and query parallelization. As the proof of the pudding is in the eating, Section 5 describes experience with the language in the PRISMA parallel database system project. This experience shows that a language with a formal basis can indeed be well used in real-world high-performance systems.

2. A sequential extended relational algebra

This section describes in short a sequential database manipulation language based on a multi-set relational algebra. The description is based on the work in [15], so the reader is refered to this publication for further details. First, the background of the language is given, and the structures of the multi-set relational model are defined. Then, as the first step in the construction of the extended relational algebra language, relational expressions are defined. Next, in step two of the construction, statements are added to the expressions to obtain a sequential database manipulation language.

2.1. *Background*

The relational data model consists of *structures*, representing the data in a database, and *operations* working on these structures. In the standard relational data model as orginally defined in [6], the structures are based on set-theory. This implies that duplicate tuples are not allowed in relations. Database practice, on the other hand, needs duplicates, both because of application semantics and because of the high processing costs associated with duplicate elimination. For these reasons, the algebra proposed here is based on multi-set theory that does allow duplicates. The choice for multi-set semantics requires a redefinition of both the relational structures (the relations) and the relational expressions. Both are defined below.

2.2. *The structures*

The structures represent the static properties of the relational data model. They have been defined originally in [6], and described thereafter in many textbooks. In this section, the structures are redefined to capture the notion of multi-sets of tuples.

The definition of multi-set relational databases is constructed below in several steps. The first basic notion is that of a *domain*.

DEFINITION 1. A *domain* $\mathcal{A}$ is a set of atomic values. The term *atomic* refers to the fact that each value in the domain is indivisible as far as operators of the relational data model are concerned.

Common types of domains are the basic data types of integer, real, boolean, and string. More specialized types as time, date, or money are possible too; note that these domains are also atomic in the sense of the definition above. In the definition below, domains are combined to form *relation schemas*.

DEFINITION 2. A *relation schema* $\mathcal{R}$ consists of a relation name and a list of attributes $\langle A_1, \ldots, A_n \rangle$. Each attribute A_i is defined on a domain $\text{dom}(A_i)$. The type of $\mathcal{R}$ is defined as $\text{dom}(\mathcal{R}) = \text{dom}(A_1) \times \cdots \times \text{dom}(A_n)$. A *relation* or *relation instance* R of relation schema $\mathcal{R}$ is a multi-set of elements in $\text{dom}(\mathcal{R})$, i.e. a function $R : \text{dom}(\mathcal{R}) \to I\!N$, where $I\!N$ denotes the domain of the natural numbers. The value of $R(x)$ is called the *multiplicity of x in R*.

A relation instance consists of *tuples* as defined below.

DEFINITION 3. A *tuple* r of schema $\mathcal{R}$ is an element in $\text{dom}(\mathcal{R})$. A tuple r is an element of relation R if its multiplicity in R is greater than zero: $r \in R \Leftrightarrow R(r) > 0$. The value of the ith attribute of tuple r is denoted as $r.i$. The number of attributes of r is denoted as $\#r$. The projection $\pi_\alpha(r)$ is obtained by concatenating the attributes from r as specified by the attribute list α into a new tuple. In this, α is a list of prefixed integers $\langle \%i_1, \ldots, \%i_n \rangle$ with $n \geq 1$ and $1 \leq i_j \leq \#r$ for $1 \leq j \leq n$. The *concatenation* of two tuples $r_1 \oplus r_2$ is defined as the concatenation of the attributes of r_1 and r_2 in the specified order. The equality of two tuples $r_1 = r_2$ is defined for tuples having the same schema; $r_1 = r_2$ holds if all corresponding attributes of r_1 and r_2 have equal values.

The operators π_α and $\oplus$ defined here on tuples are used in the sequel of this paper for relation schemas as well with obvious semantics. For reasons of brevity, their formal definition is omitted.

As stated above, relations are defined as multi-sets of tuples; this means that duplicate tuples are allowed in a relation. Multi-sets can be denoted as a collection of individual tuples r, possibly containing duplicates, or as a set of pairs $(r, R(r))$ without duplicates, where $R(r)$ denotes the number of occurrences of r in R. Further, attributes in a relation schema are ordered to enable attribute addressing by index, rather than by name. This is a notational convention that implies no restrictions with respect to the situation with explicit attribute names, but enables addressing the attributes of anonymous relations. Attribute numbers are prefixed in attribute lists to avoid ambiguity with normal integer constants.

DEFINITION 4. A *database schema* $\mathcal{D}$ is a set of relation schemas $\{\mathcal{R}_1, \ldots, \mathcal{R}_n\}$. A *database* or *database instance* D of database schema $\mathcal{D}$ is a set of relation instances $\{R_1, \ldots, R_n\}$. The set of all possible database instances of schema $\mathcal{D}$ is called the *database universe* $U_\mathcal{D}$, so $U_\mathcal{D} = \text{dom}(\mathcal{R}_1) \times \cdots \times \text{dom}(\mathcal{R}_n)$.

Note that a database schema is a set of relation schemas; consequently, relations in a database are always addressed by name. A database instance is also commonly referred to as *database state*.

2.3. Step one: basic RA expressions

This section introduces multi-set expressions on relational databases. The constructs in this algebra are based on the standard relational algebra operators as they can be found in many textbooks on database systems. Note, however, that they are modified to deal with multi-sets of tuples.

The standard relational algebra for multi-sets is defined first. This algebra contains a basic set of operators to form relational expressions. The choice of operators is closely related to the normal set-based operators. Multiple variants of the same set-based operator, like the union operators proposed in [1], are avoided. Similar to the notation for multi-set relations, the multiplicity of a tuple x in a multi-set expression E is denoted as $E(x)$.

DEFINITION 5. The *standard relational algebra* defines standard relational expressions. A database relation is a standard relational expression. Let E_1, E_2, and E_3 denote standard relational expressions; E_1 and E_2 are defined on schema $\mathcal{E}$, E_3 is defined on schema $\mathcal{E}'$. Then the following constructs are standard relational expressions:

- The *union*[1] $E_1 \uplus E_2$ collects the elements of E_1 and E_2 into a multi-set with schema $\mathcal{E}$:

$$E_1 \uplus E_2 = \{(x, E_1(x) + E_2(x)) \mid x \in \mathrm{dom}(\mathcal{E})\}$$

- The *difference* $E_1 - E_2$ "subtracts" the contents of E_2 from the contents of E_1 resulting a multi-set with schema $\mathcal{E}$:

$$E_1 - E_2 = \{(x, \max(0, E_1(x) - E_2(x))) \mid x \in \mathrm{dom}(\mathcal{E})\}$$

- The *product* $E_1 \times E_3$ forms the cartesian product of the elements of E_1 and E_3 resulting a multi-set with schema $\mathcal{E} \oplus \mathcal{E}'$:

$$E_1 \times E_3 = \{(x \oplus y, E_1(x) \cdot E_3(y)) \mid x \in \mathrm{dom}(\mathcal{E}) \wedge y \in \mathrm{dom}(\mathcal{E}')\}$$

- The *selection* $\sigma_\varphi E_1$ selects elements from a multi-set that meet a condition φ defined on individual tuples in $\mathrm{dom}(\mathcal{E})$, resulting a multi-set with schema $\mathcal{E}$:

$$\sigma_\varphi E_1 = \{(x, E_1(x)) \mid x \in \mathrm{dom}(\mathcal{E}) \wedge \varphi(x)\} \cup \{(x, 0) \mid x \in \mathrm{dom}(\mathcal{E}) \wedge \neg\varphi(x)\}$$

  In this definition, φ can be seen as a function from $\mathrm{dom}(\mathcal{E})$ into the boolean domain.
- The *projection* $\pi_\alpha E_1$ projects a multi-set E_1 on the attributes in attribute list α[2], resulting a multi-set with schema $\pi_\alpha\mathcal{E}$:

$$\pi_\alpha E_1 = \{(\pi_\alpha x, \Sigma_{\varphi(y)} E_1(y)) \mid x \in \mathrm{dom}(\mathcal{E})\}$$

  where

$$\varphi(y) \equiv y \in \mathrm{dom}(\mathcal{E}) \wedge \pi_\alpha y = \pi_\alpha x$$

- The *intersection* $E_1 \cap E_2$ produces a multi-set consisting of the elements that are both in E_1 and E_2, having schema $\mathcal{E}$:

$$E_1 \cap E_2 = \{(x, \min(E_1(x), E_2(x))) \mid x \in \mathrm{dom}(\mathcal{E})\}$$

- The *join* $E_1 \bowtie_\varphi E_2$ produces a selection on the product of E_1 and E_2, having schema $\mathcal{E} \oplus \mathcal{E}'$:

$$\begin{aligned} E_1 \bowtie_\varphi E_3 = {} & \{(x \oplus y, E_1(x) \cdot E_3(x)) \mid x \in \mathrm{dom}(\mathcal{E}) \wedge y \in \mathrm{dom}(\mathcal{E}') \wedge \varphi(x \oplus y)\} \\ & \cup \{(x \oplus y, 0) \mid x \in \mathrm{dom}(\mathcal{E}) \wedge y \in \mathrm{dom}(\mathcal{E}') \wedge \neg\varphi(x \oplus y)\} \end{aligned}$$

The intersection and join operators are not necessary from a purely functional point of view. This is shown by expressing them in the operators of the basic relational algebra below. A proof of the above equivalences can be found in [15].

$$E_1 \cap E_2 = E_1 - (E_1 - E_2)$$
$$E_1 \bowtie_\varphi E_2 = \sigma_\varphi(E_1 \times E_2)$$

The standard relational algebra above can be used for the specification of standard relational algebra expressions. The algebra lacks some important expressive power however: arithmetic expressions on attributes are not possible, duplicates cannot be removed, and aggregates over multi-sets are not included. Therefore, we extend the standard relational algebra below to include these features.

Before the extended relational algebra expressions can be defined, aggregate functions are introduced in a multi-set relational context below.

DEFINITION 6. The *multi-set aggregate functions* compute an aggregate value on a specified attribute of a multi-set expression. Let E be a multi-set defined on schema $\mathcal{E}$, and β an attribute of $\mathcal{E}$. The multi-set aggregate functions are defined as follows:

- The *count* function:

$$\text{CNT}_\beta E = \sum_{x \in \text{dom}(\mathcal{E})} E(x)$$

- The *sum* function over attribute β with a numeric domain:

$$\text{SUM}_\beta E = \sum_{x \in \text{dom}(\mathcal{E})} x.\beta \cdot E(x)$$

- The *average* function over attribute β with a numeric domain:

$$\text{AVG}_\beta E = \text{SUM}_\beta E / \text{CNT}_\beta E$$

- The *minimum* function:

$$\text{MIN}_\beta E = \min\{x.\beta \mid x \in \text{dom}(\mathcal{E}) \wedge E(x) > 0\}$$

- The *maximum* function:

$$\text{MAX}_\beta E = \max\{x.\beta \mid x \in \text{dom}(\mathcal{E}) \wedge E(x) > 0\}$$

Parameter β in the count function is a dummy parameter, included only for reasons of syntactical uniformity.

Note that the set of aggregate functions defined above is rather arbitrary; other choices can be made, including statistical aggregate functions like standard deviation for example. Note further that the average, minimum and maximum functions are in fact partial functions, since they are not defined on empty multi-sets.

DEFINITION 7. The *extended relational algebra expressions* are defined as the standard relational expressions extended with three additional constructs. Any standard relational expression is an extended relational expression. Let E be an extended relational expression defined on schema $\mathcal{E}$. Then the following constructs are extended relational expressions:

- The *extended projection* $\pi_\alpha E$ is similar to the normal projection defined above, but α contains arithmetic expressions defined on the attributes of E, rather than attributes of E only. These arithmetic expressions can be seen as functions from $\mathrm{dom}(\mathcal{E})$ into a basic domain. Given $\alpha = \langle e_1, \ldots, e_n \rangle$ with $n \geq 1$, the extended projection on a tuple x is defined as[3]:

$$\pi_\alpha x = [e_1(x), \ldots, e_n(x)]$$

Here, the square brackets denote tuple construction. Given this redefinition of the tuple projection, the definition of the extended projection operator on multi-sets is the same as the definition of the normal projection operator given before. The normal projection operator can be seen as a special case of the extended operator. The extended projection is denoted with the same symbol as the normal projection for reasons of readability; in the sequel of this paper, the π symbol denotes the extended projection.
- The *unique* expression δE calculates the multi-set obtained by duplicate removal on E, having schema $\mathcal{E}$:

$$\begin{aligned} \delta E = &\{(x, 1) \mid x \in \mathrm{dom}(\mathcal{E}) \wedge E(x) > 0\} \\ &\cup \{(x, 0) \mid x \in \mathrm{dom}(\mathcal{E}) \wedge E(x) = 0\} \end{aligned}$$

- The *groupby* expression $\Gamma_{\alpha,f,\beta}E$ on an expression E with schema $\langle A_1, \ldots, A_n \rangle$ calculates a multi-set aggregate function f on an attribute β producing a value in domain $\mathcal{F}$ per group of tuples, where the grouping is defined by equality of the attributes in the (duplicate-free) attribute list $\alpha = \langle \%a_1, \ldots, \%a_k \rangle$:

$$\Gamma_{\alpha,f,\beta}E = \{(x, 1) \mid x \in G\} \cup \{(x, 0) \mid x \in D' \wedge x \notin G\}$$

where

$$G = \left\{x \in D' \mid (\exists y \in E)(x = \pi_\alpha y \oplus [f(\sigma_{\%a_1 = x.1 \wedge \cdots \wedge \%a_k = x.k}(E), \beta)])\right\}$$

and

$$D' = \mathrm{dom}(A_{a_1}) \times \cdots \times \mathrm{dom}(A_{a_k}) \times \mathcal{F}$$

The resulting multi-set has schema $\pi_\alpha \mathcal{E} \oplus \mathit{ran}(f(\pi_\beta \mathcal{E}))$, i.e. the schema of the grouping attributes extended with the type of the range of the aggregate function. If the attribute list α is empty, the groupby expression calculates an aggregate function over the attributes of all tuples in a multi-set; in this case, the result is one single-attribute tuple:

$$\Gamma_{\langle\rangle,f,\beta}E = [f(E, \beta)]$$

2.4. *Step two: A sequential XRA*

The previous section has discussed the expressions of the extended relational algebra. In this section we add constructs that build a complete sequential data manipulation language on this basis. Note that the language including these constructs is still called an extended relational algebra, but that it is not an algebra in the mathematical meaning of the word [10].

First, the basic *statements* are introduced. The statements define constructs to be used for querying and updating a database. Statements can be grouped into *programs* to specify more complex operations against a database.

DEFINITION 8. The *extended relational algebra statements* are defined as follows. Let R be a database relation, and E an extended relational expression of the same schema. Then the following constructs are extended relational algebra statements:

- The *insert* statement $insert(R, E)$ adds the elements of E to relation R:

$$insert(R, E) \equiv R \leftarrow R \uplus E$$

- The *delete* statement $delete(R, E)$ subtracts the elements of E from relation R:

$$delete(R, E) \equiv R \leftarrow R - E$$

- The *update* statement $update(R, E, \alpha)$ modifies the elements in the intersection between R and E according to the attribute expression list α with the same schema as E:

$$update(R, E, \alpha) \equiv R \leftarrow (R - E) \uplus \pi_\alpha(R \cap E)$$

  Note that π_α is a structure-preserving extended projection operator here, i.e. it results a multi-set of the same schema as its operand.
- The *assignment* $R = E$ assigns the multi-set E to a new and implicitly defined relational variable R:

$$(R = E) \equiv R \leftarrow E$$

- The *query statement* $?E$ sends the result of expression E as output to the user of the database system; the statement has no effect on the database.

In this definition, the symbol $\leftarrow$ denotes replacement.

Extended relational algebra statements can be grouped into *programs* as defined below to specify more complex operations on a database.

DEFINITION 9. The *extended relational algebra programs* are defined as follows. Let a be an extended relational algebra statement and p an extended relational algebra program. Then the following constructs are extended relational algebra programs:

- The *single-statement program* a.
- The *multi-statement program* $p; a$.

Here, the semicolon denotes sequencing, meaning that the effects of the statements on the database and on the query output must be in the order specified by the program.

The programs as defined above can further be "encapsulated" into transactions to be able to guarantee properties like atomicity of execution and serializability with respect to concurrently executing operations. This topic is described in detail in [15].

3. A parallel extended relational algebra

This section extends the sequential extended relational algebra discussed in the previous section to a language for parallel database systems. First, the background of distributed and parallel databases is discussed, introducing the necessary concepts. Then, the relational structures of Section 2 are extended to include fragmented databases. Next, step three towards the parallel extended relational algebra is made, building on the results of steps one and two as discussed in Section 2. This step adds data distribution and collection primitives to the language. Finally, step four adds data and process allocation primitives to the language resulting from step three.

3.1. Background

This section discusses the basic concepts of distributed databases. The discussion is partially based on the work in [5].

A distributed or parallel database system consists of a number of nodes (or sites) interconnected by means of a communication network. Nodes are identified by their node number $1, \ldots, n$, where n is the number of nodes in the network. The data of a distributed database is spread over the nodes of a distributed database system. The way in which the data is split into parts is called the *data fragmentation*, the way the parts are assigned to the nodes in the system is called *data allocation.*

The fragmentation of a database determines how the global database is split up into multiple parts. Each global database relation is fragmented into one or more parts, called *relation fragments* or simply *fragments*. In general, the fragmentation of a global relation should satisfy the following properties [5]:

- The fragmentation must be complete, meaning that all data of the relation must be included in the fragments of the relation.
- The fragmentation should be disjoint: there should be no duplication in the storage of the relation data.
- The global relation must me reconstructable from the fragments, i.e. the global relation can be derived from the fragments by means of algebraic operations.

Various forms of data fragmentation are described in the literature: range-based horizontal fragmentation, hash-based horizontal fragmentation, vertical fragmentation, and mixed fragmentation [5]. In this paper, we limit ourselves to hash-based horizontal fragmentation, since this form generally enables parallel query execution best. Range-based horizontal fragmentation can be added easily, however. In the sequel of this paper, the term fragmentation refers to horizontal hash-based fragmentation.

DEFINITION 10. A *data fragmentation schema* for relation schema $\mathcal{R}$ is a pair $\text{FS}_{\mathcal{R}} = \langle \alpha, d \rangle$, where α is a duplicate-free list of attributes of $\mathcal{R}$, and $d \in \mathbb{N}^+$; α is referred to as the *fragmentation attributes* and d as the *fragmentation degree* of $\mathcal{R}$. The schema $\text{FS}_{\mathcal{R}}$ defines a set of *relation fragments* $\{R_1, \ldots, R_d\}$ for an instance R of schema $\mathcal{R}$:

$$R_i = \sigma_{\text{hash}(\alpha) \bmod d = i-1} R \quad \text{for} \quad 1 \leq i \leq d$$

Here, *hash* is a system-defined function that maps an arbitary tuple to an integer, and mod denotes the integer modulo function. The set of fragment names of a relation R is denoted

as *fragments*(R), the fragmentation attributes of R as $\alpha_f(R)$, and the fragmentation degree of R as $\delta(R)$.

The fragmentation defined above assigns each tuple of a relation R to one relation fragment R_i, dependent on the value of the attributes in α. Clearly, this fragmentation strategy satisfies the three requirements listed above, since each tuple of a relation R is assigned to a relation fragment, no tuple is assigned to more than one fragment, and relation R is obtained by taking the union of the fragments of R:

$$R = R_1 \uplus \cdots \uplus R_d$$

Each relation fragment R_i is mapped to a node of the distributed database system. This mapping is specified by a data allocation schema.

DEFINITION 11. A *data allocation schema* for a relation schema $\mathcal{R}$ and its data fragmentation schema $\langle\alpha, d\rangle$ is a list $\mathrm{AS}_{\mathcal{R}} = \langle a_1, \ldots, a_d\rangle$, where $a_i \in \mathbb{N}^+$. Schema $\mathrm{AS}_{\mathcal{R}}$ defines a mapping of each fragment R_i of an instance R of schema $\mathcal{R}$ to a node a_i of a distributed database system.

3.2. *The structures*

Given the fragmentation and allocation concepts introduced above, distributed relations and distributed databases can be defined as follows.

DEFINITION 12. A *distributed relation schema* is a triple $\mathcal{R}^D = \langle\mathcal{R}, \mathrm{FS}_{\mathcal{R}}, \mathrm{AS}_{\mathcal{R}}\rangle$, where $\mathcal{R}$ is a relation schema as defined in Section 2, $\mathrm{FS}_{\mathcal{R}}$ is a fragmentation schema, and $\mathrm{AS}_{\mathcal{R}}$ an allocation schema. A *distributed relation instance* or *distributed relation* is a set of fragments defined by $\mathrm{FS}_{\mathcal{R}}$ and allocated as specified by $\mathrm{AS}_{\mathcal{R}}$.

DEFINITION 13. A *distributed database schema* $\mathcal{D}^D$ is a set of distributed relation schemas $\{\mathcal{R}_1^D, \ldots, \mathcal{R}_n^D\}$. A *distributed database instance* or simply *distributed database* is a set of distributed relation instances.

EXAMPLE 1. As an example, we use a database describing beers and brewers, having the following simple global (unfragmented) schema:

$$\begin{array}{c} \mathit{Beer}\ (\mathit{Name}, \mathit{Brewer}, \mathit{Alcperc}) \\ \mathit{Brewer}\ (\mathit{Name}, \mathit{City}, \mathit{Country}) \end{array}$$

Assume that we have a database system with 5 nodes. Now we can fragment relation *beer* into 3 fragments based on attribute *name* and allocate these fragments on nodes 1 to 3. Relation *brewer* can be fragmented in two fragments based on attribute *name* and allocated to nodes 4 and 5. So we have a distributed database schema consisting of the following distributed relation schema's:

$$\begin{aligned} \mathit{Beer}^D &= \langle \mathit{Beer}, \langle \mathit{Name}, 3\rangle, \langle 1, 2, 3\rangle\rangle \\ \mathit{Brewer}^D &= \langle \mathit{Brewer}, \langle \mathit{Name}, 2\rangle, \langle 4, 5\rangle\rangle \end{aligned}$$

3.3. *Step three: A parallel XRA with data distribution*

To perform parallel data processing in a database system, data to be processed must be available at multiple processing nodes. Two situations can occur here: either the data is already spread over the right nodes, or it is not and has to be distributed for processing. For the last situation, two data distribution primitives are added to the extended relational algebra.

DEFINITION 14. The *extended relational algebra statements* as defined in Definition 8 are extended as follows. Let E denote an arbitrary extended relational algebra expression as defined before. Then the following constructs are extended relational algebra statements:

- The *copy* statement $copy(E, T_1, \ldots, T_n)$ assigns the multi-set E to each of the new and implicitly defined relational variables T_i in a parallel fashion:

$$copy(E, T_1, \ldots, T_n) \equiv \begin{cases} T_1 = E, \\ T_2 = E, \\ \vdots \\ T_n = E \end{cases}$$

- The *split* statement $split(E, a, T_1, \ldots, T_n)$ distributes the elements of the multi-set E over the new and implicitly defined relational variables T_i in a parallel fashion, where the assignment of individual tuples t to one of the variables is determined by the value $hash(\pi_\alpha t)$:

$$split(E, \alpha, T_1, \ldots, T_n) \equiv \begin{cases} T_1 = \sigma_{\text{hash}(\alpha)\text{mod}n=0} E, \\ T_2 = \sigma_{\text{hash}(\alpha)\text{mod}n=1} E, \\ \vdots \\ T_n = \sigma_{\text{hash}(\alpha)\text{mod}n=n-1} E \end{cases}$$

The punctuation by means of commas between statements in the extended relational algebra programs above is used to indicate simultaneous execution.

To collect the data from multiple fragments of relations or intermediate results of operations, a *multi-union* is added to the expressions of the extended relational algebra.

DEFINITION 15. Let $E_1, E_2, \ldots, E_n$ denote relational expressions (see Definitions 5 and 7) defined on schema $\mathcal{E}$. Then the following construct is a relational expression:

- The *multi-union* expression[4] $\uplus(E_1, E_2, \ldots, E_n)$ collects the elements of $E_1, E_2, \ldots$ into a multi-set with schema $\mathcal{E}$:

$$\uplus(E_1, E_2, \ldots, E_n) = \{(x, E_1(x) + E_2(x) + \cdots + E_n(x)) \mid x \in \text{dom}(\mathcal{E})\}$$

The introduction of the *copy*, *split*, and *multi-union* operators allows the easy construction of rooted directed acyclical graphs, the typical structures necessary for parallel query evaluation on fragmented databases (as opposed to trees for sequential query evaluation on non-fragmented databases). The operators have the function of *send* and *receive* operations

in message-passing languages like PFAD [16], but are defined on a higher level of abstraction. An example illustrating the use of the operators defined above follows in the sequel of this paper.

3.4. Step four: A parallel XRA with process allocation

In the sections above, various types of operators have been defined that are to be used for parallel data processing. These operators will be executed by processes in a parallel database system. To obtain the best parallel execution, it must be possible to specify on which processors these processes must be executed. In this way, minimal data movement and processor contention can be reached. For this purpose, allocation pragma's are introduced that can be attached to the various operators.

DEFINITION 16. An *allocation pragma* is either an absolute or a relative allocation pragma. An *absolute allocation pragma* is defined as a prefixed integer αi with $i \in \mathbb{N}$. A *relative allocation pragma* is defined as a prefixed integer ρi with $i \in \mathbb{N}$. Here $\mathbb{N}$ denotes the domain of the natural numbers.

Allocation pragma's can be defined to specify allocated relational expressions and statements.

DEFINITION 17. An *allocated relational expression* is defined as $[E]_a$, where E is a relational expression as defined in Definitions 5, 7, and 15, with the exception of the relational constant, and a is an allocation pragma. If a is a relative allocation pragma ρi, then $0 \leq i \leq n$, where n is the number of operands of the top-level operator in E.

The meaning of an allocated expression $[E]_a$ is that the top-level relational operation in E must be executed at the node indicated by a in the following way:

- an absolute pragma αi with $i > 0$ indicates the node with number $((i-1) \text{ MOD } n)+1$, where n is the number of nodes in the system;
- an absolute pragma $\alpha 0$ indicates a don't care, so any node in the system is fine;
- a relative pragma ρi with $i > 0$ indicates the node on which the ith operand of the top-level expression of E is located or executed;
- a relative pragma $\rho 0$ indicates any node on which none of the operands of the top-level expression of E is located or executed.

So, absolute pragma's allow process allocation with respect to the machine architecture, relative pragma's allow process allocation with respect to the expression structure and the allocation of database fragments.

DEFINITION 18. An *allocated relational statement* is defined as $[S]_a$, where S is a *copy* or *split* statement as defined in Definition 14 and a is an allocation pragma. If a is a relative allocation pragma ρi, then $i \in \{0, 1\}$.

The meaning of an allocated *copy* or *split* statement $[S]_a$ is that the operation of S must be executed at the node indicated by a. The meaning of a is the same as described above for allocated expressions. Note that relative allocation is with respect to the input operand of

the *copy* or *split* statement. Allocation of *insert*, *delete*, and *update* statements is for obvious reasons determined by the allocation of the fragments to be modified by the statements.

4. Rewriting expressions and statements

The use of an algebra-based language enables the use of high-level rewrite rules for query optimization. These rules express equivalence transformations that can be applied to algebra expressions in order to obtain expressions that produce the same results at a lower execution cost. Table 1 shows a number of examples of these query optimization rules, as they can be found in many textbooks (see e.g. [5, 18]).

Most of the equivalences from the set-based theory also hold in the multi-set context, but not all of them. An example is the idempotency of the union operator, which holds in the set-based theory, but doesn't hold in the multi-set context:

$$R \uplus R \neq R \quad \text{if } R \neq \emptyset$$

On the other hand, the multi-set algebra allows rewrite rules that are not correct in a set-based algebra. An example is the following rule:

$$\Gamma_{\alpha,\mathrm{CNT},\beta} R \rightarrow \Gamma_{\alpha,\mathrm{CNT},\gamma} \pi_\alpha R \qquad \text{if } \gamma \in \alpha$$

In a set-based context, the projection operator must remove duplicates after attribute removal and may thus influence the result of the count function. A useful application of the above rewrite rule is discussed in Section 5.2.

As the extension to the parallel environment presented in this paper is also algebra-based, rewrite mechanisms can also be used for the fragmentation and parallelization of queries. A number of example rules is shown in Table 2. These rules assume that relation R is fragmented into m fragments, and relation S into n fragments.

The parallelization of expressions with a binary operator may lead to constructs that do not have the form of an expression tree, but of a directed acyclic graph (DAG). Consequently, the rewriting does not result a single expression, but an extended relational algebra program with data distribution operators. The rewrite rule for the join in Table 2 is an example of this.

Note that the parallelization of aggregation operators as shown in rule 4 in Table 2 requires an algebra with multi-set semantics to be correct. A set-based algebra would eliminate duplicate tuples in the intermediate results between the inner and outer group-by operators.

Table 1. Query optimization rewrite rules.

1a	$\sigma_\varphi(R \bowtie_\psi S) \rightarrow \sigma_\varphi R \bowtie_\psi S$	if $attr(\varphi) \subseteq attr(R)$
1b	$\sigma_\varphi(R \bowtie_\psi S) \rightarrow R \bowtie_\psi \sigma_\varphi S$	if $attr(\varphi) \subseteq attr(S)$
2a	$\pi_\alpha(R \bowtie_\varphi S) \rightarrow \pi_\alpha R \bowtie_\varphi S$	if $attr(\alpha) \subseteq attr(R) \wedge attr(R) \cap attr(\varphi) \subseteq attr(\alpha)$
2b	$\pi_\alpha(R \bowtie_\varphi S) \rightarrow R \bowtie_\varphi \pi_\alpha S$	if $attr(\alpha) \subseteq attr(S) \wedge attr(S) \cap attr(\varphi) \subseteq attr(\alpha)$
3	$\sigma_\varphi(R \times S) \rightarrow R \bowtie_\varphi S$	
4	$\sigma_\varphi \sigma_\psi R \rightarrow \sigma_{\varphi \wedge \psi} R$	
5	$R \bowtie_\varphi (S \bowtie_\psi T) \rightarrow (R \bowtie_\varphi S) \bowtie_\psi T$	

Table 2. Query parallelization rewrite rules.

0	$R \rightarrow \left[\biguplus_{i=1}^{i=m} R_i\right]$	
1	$\sigma_\varphi R \rightarrow \left[\biguplus_{i=1}^{i=m} [\sigma_\varphi R_i]_{\rho 1}\right]_{\rho 0}$	
2	$\pi_\alpha R \rightarrow \left[\biguplus_{i=1}^{i=m} [\pi_\alpha R_i]_{\rho 1}\right]_{\rho 0}$	
3	$\delta R \rightarrow \left[\biguplus_{i=1}^{i=m} [\delta R_i]_{\rho 1}\right]_{\rho 0}$	
4a	$\Gamma_{\alpha,f,\beta} R \rightarrow \left[\Gamma_{\alpha,f,\beta'} \biguplus_{i=1}^{i=m} [\Gamma_{\alpha,f,\beta} R_i]_{\rho 1}\right]_{\rho 0}$	if $f \in$ {CNT, SUM, MIN, MAX}
4b	$\Gamma_{\alpha,f,\beta} R \rightarrow \left[\Gamma_{\alpha,f,\beta} \biguplus_{i=1}^{i=m} R_i\right]_{\rho 0}$	if $f \in$ {AVG}
5	$R \bowtie_\varphi S \rightarrow \begin{cases} split(R_1, \alpha, t_{1,1}, \ldots, t_{1,n}); \\ \vdots \\ split(R_m, \alpha, t_{m,1}, \ldots, t_{m,n}); \\ \left[\biguplus_{i=1}^{i=n} \left[\left[\biguplus_{j=1}^{j=m} t_{j,i}\right] \bowtie_\varphi S_i\right]_{\rho 2}\right]_{\rho 0} \end{cases}$	

EXAMPLE 2. As an example we use the beer and brewer database presented before. Assume we require an overview of the names of all countries in the database in which strong beers are brewn, i.e. beers with more than 8 percent alcohol. In a non-fragmented database, this would be the following query:

$$\delta\pi_\alpha\sigma_\varphi(Beer \bowtie_\psi Brewer)$$

with

$$\alpha = Brewer.Country$$
$$\varphi = (Beer.Alcperc > 8)$$
$$\psi = (Beer.Brewer = Brewer.Name)$$

Using optimization rule 1a from Table 1, we can rewrite this query into the following equivalent query:

$$\delta\pi_\alpha(\sigma_\varphi Beer \bowtie_\psi Brewer)$$

Given the distributed database schema of Example 1, we can rewrite this query into a parallel extended relational algebra program using the rewrite rules in Table 2. Assuming that we don't want to redistribute relation *Beer*, this query can be rewritten using rules 1 and 5 to obtain the following program:

$$\begin{aligned}
&[split(Brewer_1, \langle Name\rangle, t_{11}, t_{12}, t_{13})]_{\rho 1};\\
&[split(Brewer_2, \langle Name\rangle, t_{21}, t_{22}, t_{23})]_{\rho 1};\\
&t_{31} = [\sigma_\varphi Beer_1]_{\rho 1};\\
&t_{32} = [\sigma_\varphi Beer_2]_{\rho 1};\\
&t_{33} = [\sigma_\varphi Beer_3]_{\rho 1};\\
&t_{41} = [t_{31} \bowtie_\psi \uplus(t_{11}, t_{21})]_{\rho 1};\\
&t_{42} = [t_{32} \bowtie_\psi \uplus(t_{12}, t_{22})]_{\rho 1};\\
&t_{43} = [t_{33} \bowtie_\psi \uplus(t_{13}, t_{23})]_{\rho 1};\\
&?[\delta[\pi_\alpha[\uplus(t_{41}, t_{42}, t_{43})]_{\alpha 1}]_{\rho 1}]_{\rho 1}
\end{aligned}$$

In the above program, the fragments of relation *Brewer* are refragmented to fit the fragmentation of relation *Beer*, so three local joins can be used at the location of the *Beer* fragments. If the result of the multi-union in the last statement is expected to be large, this statement can be further parallelized using rewrite rules 2 and 3 from Table 2 to obtain the following:

$$t_{51} = [\delta[\pi_\alpha t_{41}]_{\rho 1}]_{\rho 1};$$
$$t_{52} = [\delta[\pi_\alpha t_{42}]_{\rho 1}]_{\rho 1};$$
$$t_{53} = [\delta[\pi_\alpha t_{43}]_{\rho 1}]_{\rho 1};$$
$$?[\delta \uplus (t_{51}, t_{52}, t_{53})]_{\alpha 1}$$

Note that the statements of a parallel program like the one above can be executed in a parallel fashion, provided that the sequential semantics of the effects on the database and the query output are guaranteed [12]. Parallelism can include both parallel execution of independent statements (*horizontal parallelism*), and parallel execution of statements that have a producer-consumer relationship (*vertical parallelism*). In the second program in the above example, horizontal parallelism can exist between the first three statements, and vertical parallelism between each of the first three statements and the fourth statement.

Parallelization is (of course) not limited to query statements only. Statements to modify a fragmented database can be parallelized as well, as shown in the following example.

EXAMPLE 3. Suppose we want to insert the contents of a relation *NewBeers* into the *Beers* relation. In a non-fragmented database this would be accomplished by the following statement:

$$insert(Beers, NewBeers)$$

Given the distributed database schema of Example 1 and assuming that relation *NewBeers* is not fragmented, we can rewrite the above statement into the following parallel program:

$$[split(NewBeers, \langle Name \rangle, t_{11}, t_{12}, t_{13})]_{\rho 1};$$
$$insert(Beer_1, t_{11});$$
$$insert(Beer_2, t_{12});$$
$$insert(Beer_3, t_{13})$$

5. The practice: PRISMA/DB

A dialect of the extended relational algebra presented in this paper has been used as the main database manipulation language (DML) in the PRISMA/DB parallel main-memory database system [2]. This language, called XRA, is described in detail in [11] and includes a number of constructs not covered by this paper for reasons of brevity, such as operators for range-based data fragmentation and recursively defined expressions, and transaction brackets (*begin* and *commit*).

This section describes the use of the algebra-based XRA language in the PRISMA/DB system in short; some issues are covered more extensively in [2]. First, an overview of

the use of XRA in PRISMA/DB is given. Next, attention is paid to query rewriting in PRISMA/DB. Finally, the execution of XRA constructs is described.

5.1. *XRA in PRISMA/DB*

In PRISMA/DB, the XRA language has been used both as an external DML to express user queries and transactions in, and as an internal means of communication between the various modules of PRISMA/DB [2]. This algebra-based approach throughout the system has resulted in a high degree of expressiveness and flexibility, opening ways to both complete functionality including e.g. integrity control [13] and high performance through parallel query execution [19].

Figure 1 depicts the (simplified) basic architecture of PRISMA/DB [2]. It consists of a user interface (UI), query compiler (QC), query optimizer (QO), transaction manager (TM), data manager (called one-fragment-manager in PRISMA/DB, hence OFM), data dictionary (DD), and concurrency controller (CC).

The user at the user interface can either use XRA as data manipulation language, or choose SQL or PRISMAlog (a logic query language like DataLog). As PRISMA/DB offers complete fragmentation and location transparency [5], all queries are stated in terms of global relations. So users never need to be aware of the constructs introduced in Section 3. These are generated by the system as explained below.

The query compiler module translates all queries in languages other than XRA into XRA. This allows XRA to be further used as interface language in the system. It is so used between query compiler and query optimizer, between query optimizer and transaction manager, and between transaction manager and data manager. Further, for integrity control purposes, integrity constraints are stored in XRA format in the data dictionary for use by the transaction manager [13].

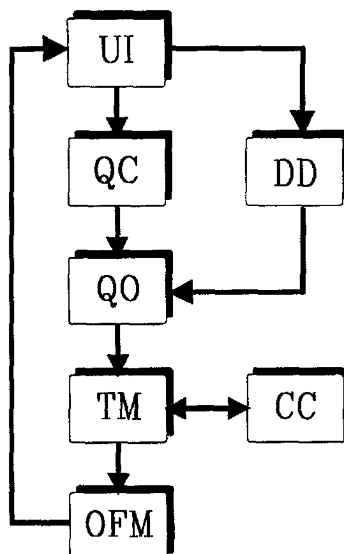

Figure 1. PRISMA/DB basic architecture.

5.2. XRA rewriting

After being processed by the query compiler, queries in XRA are processed by the query optimizer. This module has two main tasks: optimizing a given query to an equivalent query that can be executed at lower cost, and resolving fragmentation transparency. The latter is performed by replacing global relations by relation fragments and operations on relations by operations on relation fragments (both for base relations and intermediate query results).

PRISMA/DB employs the data fragmentation and allocation schemes and the data redistribution operators described in the previous section to obtain flexible mechanisms for the use of parallelism in query execution [2, 19]. This means that the degree of parallelism can be specified per relational operator, as opposed to approaches where the degree of parallelism is fixed by the system architecture, e.g. as in the GAMMA database machine [8]. In optimizing fragmented queries, the query optimizer uses relative allocation pragma's to control inter-node data transport. It needs not be concerned with physical data allocation.

Due to the algebra-based approach, both the optimization and parallelization tasks of the query optimizer can be performed by the same algebra rewrite engine. For this purpose, the engine is equipped with a set of query optimization rules and a set of query fragmentation rules, as they are discussed in the previous section. The use of a single algebra rewrite engine results in a high-level design of the query optimizer, and flexible optimization strategies.

The use of a multi-set model allows for certain optimization transformations that cannot be safely performed in a set-based environment. An important example is the insertion of projection operators in query trees to decrease the amount of data transport between nodes in the parallel machine. In the multi-set case, the projection operator does not affect the cardinality of its operand, but in the set-based case it does (as a consequence of duplicate removal). This is illustrated in the example below.

EXAMPLE 4. Using the example database schema introduced before, we want to compute the number of beers brewn outside the Netherlands. This can be accomplished by means of the following query:

$$\Gamma_{\langle\rangle,\mathrm{CNT},\mathrm{Beer.Name}}(\mathrm{Beer} \bowtie_{\mathrm{Brewer}=\mathrm{Name}} \sigma_{\mathrm{Country}\neq\text{“NL”}}\mathrm{Brewer})$$

Now suppose that the join operations are to be executed on the nodes where the *Brewer* relation resides. This means that the tuples of the *Beer* relation have to be transported to these nodes. To compute the query, only the attribute *Brewer* of relation *Beer* is necessary. This means that we can optimize the query to save on data communication costs by inserting a projection operation that removes irrelevant attributes. The resulting query is the following:

$$\Gamma_{\langle\rangle,\mathrm{CNT},\mathrm{Beer.Brewer}}(\pi_{\mathrm{Brewer}}\mathrm{Beer} \bowtie_{\mathrm{Brewer}=\mathrm{Name}} \sigma_{\mathrm{Country}\neq\text{“NL”}}\mathrm{Brewer})$$

In the set-based model, the projection operation would affect the cardinality of its operand (assuming there are brewers that brew more than one beer type), thus causing an incorrect query result.

In a set-based environment, duplicate removal operations are necessary after each operation that may produce duplicate tuples. If the data is fragmented (as usually the case in a parallel database system), costly inter-node data comparison may be necessary. Clearly, this is not desirable in a high-performance parallel environment.

5.3. *XRA execution*

After being rewritten, queries are sent to the transaction manager, which controls query execution. This module has among others the following tasks: resolving location transparency such that data managers can be addressed at the various nodes of a parallel system, and scheduling of parallel query execution. Resolving location transparency is performed by replacing relative allocation pragma's in the XRA programs by absolute allocation pragma's. This process is performed by analyzing the programs starting with the fixed locations of the fragments of the base relations.

The algebra-based approach enables easy analysis of XRA programs for high-level graph-based scheduling as described in [12]. This scheduling is necessary to be able to execute multiple XRA statements in paralel without violating the sequential semantics of the program. This approach avoids the inclusion of detailed scheduling information within the XRA programs.

Vertical parallelism in query execution is enabled by inserting *channels* for tuple transport in a pipelined fashion between operations that have a producer-consumer relationship. This is both the case for operations within one statement and operations in different statements. In the last case, the channel replaces the relational variable shared by both operations.

6. Conclusions

This paper presents a formal approach to the construction of a multi-set extended relational algebra for parallel database systems with fragmented relations. This approach has a firm theoretical background with a close connection to the standard relational algebra, ensuring a clear semantics of the developed language. The approach has also a close connection to the practice of parallel database systems, as it can handle fragmentation and parallelism in query processing and is defined in terms of multi-sets, as required by most practical environments.

The algebra-based approach as described in this paper provides a good basis for expression rewriting that can be used for query optimization and query parallelization. The multi-set model allows for certain optimization transformations that cannot be safely applied in set-based environments.

The experience in the PRISMA project has proven that the approach is a viable one. The PRISMA/DB parallel main-memory database system uses a dialect of the language described in this paper as its main database manipulation language. The use of the language has been one of the factors in creating a system with both flexible parallel query execution strategies and a high performance. The performance of PRISMA/DB has proven to be in the range of the fastest parallel database machines available. A detailed performance evaluation is described in [19].

Acknowledgments

Annita Wilschut and Carel van den Berg are acknowledged for their contributions to the design and implementation of the XRA language in PRISMA/DB. Rolf de By is thanked for his help with the formal aspects of multi-sets.

Notes

1. The $\uplus$ symbol is used here for the multi-set union to avoid confusion with the set union, denoted by the usual symbol $\cup$. Both types of unions are used in this paper; for other operators, only the multi-set type is used.
2. Here the summation $\Sigma_{\varphi(x)} f(x)$ is to be interpreted as the sum of $f(x)$ for all x satisfying $\varphi(x)$.
3. As for the normal projection, the extended projection operator is used for relation schemas as well.
4. Note that we use the same symbol for the standard two-operand union and the multi-union. This overloading does not cause any problems, since the syntax of expressions clearly indicates which operator is meant.

References

1. J. Albert, "Algebraic Properties of Bag Data Types," Proceedings 17th International Conference on Very Large Data Bases, Barcelona, Spain, 1991.
2. P.M.G. Apers, C.A.v.d. Berg, J. Flokstra, P.W.P.J. Grefen, M.L. Kersten, and A.N. Wilschut, "PRISMA/DB: A Parallel, Main-Memory Relational DBMS," IEEE Transactions on Knowledge and Data Engineering, vol. 4, no. 6, 1992.
3. H. Boral et al., "Prototyping Bubba, a Highly Parallel Database System," IEEE Transactions on Knowledge and Data Engineering, vol. 2, no. 1, 1990.
4. K. Bratbergsengen, "Relational Algebra Operations," Proceedings Workshop on Parallel Database Systems, Noordwijk, The Netherlands, 1990.
5. S. Ceri and G. Pelagatti, Distributed Databases, Principles and Systems, McGraw-Hill, New York, USA, 1984.
6. E.F. Codd, "A Relational Model for Large Shared Data Banks," Communications of the ACM, vol. 13, no. 6, 1970.
7. H. Darwen and C.J. Date, "The Third Manifesto," SIGMOD Record, no. 3, 1995.
8. D.J. DeWitt et al., "The GAMMA Database Machine Project," IEEE Transactions on Knowledge and Data Engineering, March 1990.
9. D. DeWitt and J. Gray, "Parallel Database Systems: The Future of High Performance Database Systems," Communications of the ACM, vol. 35, no. 6, 1992.
10. A. Gill, Applied Algebra for the Computer Sciences, Prentice-Hall, Englewood Cliffs, USA, 1976.
11. P.W.P.J. Grefen, A.N. Wilschut, and J. Flokstra, PRISMA/DB 1.0 User Manual, Memorandum INF91-06, University of Twente, The Netherlands, 1991.
12. P.W.P.J. Grefen and P.M.G. Apers, "Dynamic Action Scheduling in a Parallel Database System," Procs. Conf. on Parallel Architectures and Languages Europe 1992, Paris, France, 1992.
13. P.W.P.J. Grefen, "Combining Theory and Practice in Integrity Control: A Declarative Approach to the Specification of a Transaction Modification Subsystem," Procs. 19th Int. Conf. on Very Large Data Bases, Dublin, Ireland, 1993.
14. P.W.P.J. Grefen and P.M.G. Apers, "Integrity Control in Relational Database Systems—An Overview," Data and Knowledge Engineering, North-Holland. vol. 10, no. 2, 1993.
15. P.W.P.J. Grefen and R.A. de By, "A Multi-Set Extended Relational Algebra—A Formal Approach to a Practical Issue," Procs. 10th Int. Conf. on Data Engineering, Houston, Texas, USA, 1994.
16. B.E. Hart, S. Danforth, and P. Valduriez, "Parallelizing a Database Programming Language," Procs. Int. Symp. on Databases in Parallel and Distributed Systems, Austin, Texas, USA, 1988.
17. A. Klausner and N. Goodman, "Multirelations—Semantics and Languages," Proceedings 11th International Conference on Very Large Data Bases. Stockholm, Sweden, 1985.
18. J.D. Ullman, Principles of Database Systems, Second Edition, Computer Science Press, Rockville, USA, 1982.
19. A.N. Wilschut, J. Flokstra, and P.M.G. Apers, "Parallelism in a Main-Memory System: The Performance of PRISMA/DB," Proceedings 18th International Conference on Very Large Data Bases. Vancouver, Canada, 1992.