Bag Relational Algebra with Grouping and Aggregation over C-Tables with Linear Conditions

Lubomir Stanchev
Computer Science Department
Indiana University - Purdue University Fort Wayne
Fort Wayne, IN, USA
stanchel@ipfw.edu

***Abstract*—We introduce bag relational algebra with grouping and aggregation over a particular representation of incomplete information called c-tables, which was first introduced by Grahne in 1984. In order for this algebra to be closed and "well-defined", we adopt the closed world assumption as described by Reiter in 1978 and extend the tuple and table conditions to linear ones. We explore the problem of rewriting and simplifying this novel type of c-tables, show how to perform equivalence test for c-tables, and argue why it is difficult to create a canonical form for c-tables. We present certain answer semantics for a full-blown relational algebra with grouping and aggregation and accordingly present algorithms for executing the different relational algebra operators over our representation of incomplete information. The algorithms run in polynomial time relative to the size of the precise information, which makes them a candidate for implementation as part of a DBMS engine that supports storage and retrieval of incomplete information.**

Keywords-incomplete information*; *c-tables*; *relational model*; *null values*; *bag semantics

I. INTRODUCTION

This paper extends a conference paper on the topic of querying incomplete information ([1]). We have added theoretical results on simplifying and checking the equivalence of c-tables and discussion on the existence of a canonical form for c-tables. We have also expanded the description of all algorithms that implement non-trivial relational algebra operators, such as monus, grouping, and aggregation, and added detailed proofs on the correctness and time complexity to all algorithms.

Many times, when information is entered into databases, the values for some of the fields are left empty for various reasons. In some cases, partial information about the blank fields is available. However, existing relational database technology does not allow for such information to be processed. Imielinski and Lipski in [2] were among the first to propose richer semantics for null values that allows for incomplete information to be processed. However, their model was based on set semantics. Later on, Libkin and Wong published a paper on querying incomplete information in databases with multisets ([3]), but included only a limited set of operators that excluded grouping and aggregation. Other papers that tackle the problems of storing and querying incomplete information include [4], [5], [6], [7], [8]. However, they all fail to explore grouping and aggregation over bag semantics.

In this paper, we fill a gap in published research in the area of storing and querying incomplete information. More precisely, we show how bag relational algebra with grouping and aggregation can be applied over incomplete information represented as a particular variation of *c-tables*. A c-table consists of a of *c-tuples* and a *global condition*, where every c-tuple contains a regular tuple that may include variables for some of its fields plus a local condition (See Table I for an example). The semantics of a c-table is determined by the set of relational tables that it represents, where each representation is derived from a valuation for the variables in the c-table. In order for the relational algebra over c-tables to be closed and *well defined*, we define the semantics of a c-table to be over the *closed world assumption*, as defined in [9], and we extend local and global conditions to be linear. We will refer to such c-tables as *linear c-tables*, where the exact semantics will be presented in Section II-A.

C-tables were first introduced by Grahne in [10] to have local and global conditions that did not contain the "+" operator and the ">" relation. Later on, Grahne added the ">" relation in [4]. However, we are not aware of any published research that allows for the "+" operator to be part of the local or global condition of a c-table. On the other hand, introducing the "+" operator is required in order for relational algebra with aggregation over c-tables to be closed.

Note that several different linear c-tables may have the same semantics, that is, have the same set of representations. This is the reason why it is desirable to be able to check for equivalence between linear c-tables and be able to normalize linear c-tables. For example, when we store or visualize a linear c-table, we would want to use a compact and easy to understand representation. In the paper we present a novel procedure for simplifying linear c-tables that runs in polynomial time relative to the size of the precise information. We show why it is difficult to construct a canonical form for linear c-tables and solve the problem of comparing linear

c-tables for equality.

The main contributions of the paper are the algorithm for simplifying linear c-tables, the algorithm for comparting two c-tables for equality, and the algorithms for performing the different relational operators over linear c-tables. While the implementation of the operator projection, selection, and inner join are similar to the case of set semantics (see [2]), the algorithms for monus, duplicate elimination, grouping, and aggregation are non-trivial and novel.

A. Motivation

Real world requirements have shown the importance of storing and querying incomplete information. However, contemporary database management systems (DBMSs) provide only limited support (that is, only null values). Part of the reason is the lack of research in the area. While the problem of storing incomplete information is somewhat solved, querying incomplete information remains an open research challenge. This paper makes a significant step towards solving the later problem.

The main hurdle towards the implementation of a DBMS that can processes rich incomplete information is the intrinsic high cost of managing such information. However, note that the algorithms that we present for performing the various relational algebra operators are non-polynomial relative only to the size of the incomplete information. Taking into account the ever-increasing speed of computational resources, we believe that incorporating tools that store and query incomplete information within commercial database engines is feasible and practical. This work can play a key part in such an endeavor. For example, since the code for executing bag relational algebra operators is an important part of the kernel of a SQL engine, our algorithms can be used to implement a SQL engine that can query incomplete information stored as linear c-tables.

In what follows, in Section II we define a representation of incomplete information in terms of linear c-tables. In Section III we describe how linear c-tables can be simplified and compared for equality and explore the problem of existence of canonical form for c-tables. In Section IV we define bag relational algebra operators over linear c-tables and present example algorithms for their implementation. In Section V the problems of grouping and aggregation over linear c-tables are explored. Section VI provides a summary or the presented work and addresses areas for future research.

II. C-Tables with Linear Conditions

The problem of representing incomplete information in the relational model is almost as old as the relational model itself ([11], [12], [13], [14], [15]). When a null value appears in a relational table, its value can be interpreted as no information available, only partial information available, value not applicable, and so on. Most of the research on null values has concentrated on the first two meanings. Known representations of relational tables adapting these meanings for nulls include Codd tables, naïve tables, Horn tables and c-tables. Codd tables are relational tables, where the values of some the fields can be null. Naïve tables are an extension of Codd tables, where each null is given a label and nulls having the same label represent the same unknown value. C-tables are naïve tables with a local condition associated which each c-tuple and a single global condition associated with each c-table. A c-tuple in a c-table is part of the representation of the c-table under some valuation when the local condition of the c-tuples and global condition of the c-table are both true. Horn tables are a special kind of c-tables in which the local and global conditions are restricted to Horn clauses.

Grahne, in [4], considered Boolean conditions over the system $\langle R, \{>, =\}\rangle$ (i.e., Boolean expressions with variables and constants defined over the set R extended with "$>$" and "$=$"). To the best of our knowledge, except for [1], this is the most expressive system for expressing c-table conditions in published research.

In this paper we explore c-tables with conditions over the system $\langle \mathbb{R},\ \{>,\ =,\ +\}\rangle \cup \langle \mathbf{S}, \{=, \neq\}\rangle$, where $\mathbb{R}$ is used to denote the set of real numbers and $\mathbf{S}$ is the set of strings over some finite alphabet. While the "$+$" operator is introduced in order to make the algebra closed relative to aggregation, the system over strings is introduced in order to extend the expressive power of c-tables. Note that we do not explore conditions over $\langle \mathcal{Z},\ \{>,\ =,\ +\}\rangle$, where $\mathcal{Z}$ is the set of integers, or over $\langle \mathbb{R},\ \{>,\ =,\ *,\ +\}\rangle$. The reason is that, although these systems are more expressive, reasoning with them is much harder. For example, Fischer and Rabin have shown that the time complexity of deciding whether a formula over the first system is satisfiable is super exponential ([16]). Similarly, the time complexity of the fastest known algorithm for solving the same problem for the second system, which is presented in [17], is higher than exponential.

A. Definitions

We next present the syntax and semantics of a linear c-table.

Definition 1 (syntax of linear c-table): A *linear c-table* T as a finite and unordered bag of *linear c-tuples* and a *global condition*[1]. A linear c-tuple with attributes $\{A_i\}_{i=1}^{a}$ is the sequence of mappings from A_i to $D(A_i) \cup V_i$ plus a local condition, where i ranges from 1 to a, $D(A_i)$ denote the domain of A_i and V_i is used to represent a possibly infinite but countable set of variables over $D(A_i)$. The local and global conditions can range over the system $\langle \mathbb{R}, \{>, =, +\}\rangle \cup \langle \mathbf{S}\ \{=, \neq\}\rangle$.

[1]In order to keep the notation simple, we do not use special syntax for c-tuples and c-tables, where it will be clear from the context when we are referring a c-table (c-tuple) and when to a relational table (tuple).

name	*school*	*condition*
John	y	$x = 1$
Mark	y	$x \neq 1$
q	z	TRUE

g.c. $(q \neq$"Mark"$) \wedge (q \neq$"John"$) \wedge (z \neq y)$

Table I
AN EXAMPLE LINEAR C-TABLE

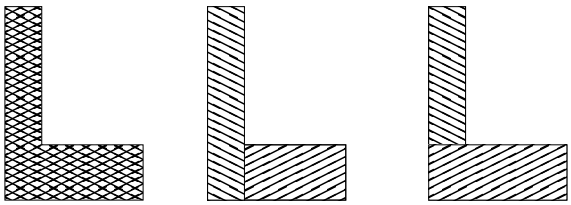


Figure 1. Three different ways to represent the same two-dimensional point set as union of polyhedra

Table I shows an example of a linear c-table. We will refer to the part of a linear c-table where the data is stored as the *main part* and to the remaining parts as the *local condition part* and the *global condition part*, respectively. In Table I, x, y, z and q are used to represent variables. Since our model is limited only to the domains of real numbers and strings, the domain of a variable that does not appear in the main part of a linear c-table can be inferred from the context in which it appears. Fore example, we can use the local condition $x = 1$ to deduce that the domain of x is the set of real numbers.

Table I expresses the information that either there are no students or there are two students that study in different schools and the name of one of them is "John" or "Mark" and the name of the other one is neither "John" nor "Mark". Note that in this example and throughout the paper we will be using the closed world assumption. The assumption states that the database contains all existing individuals. In our example, we have used this assumption to conclude that there are at most two students in the database.

In order to formally define the semantics of a c-table, Imielinski and Lipski introduce a function called *rep* that maps a c-table T to a possibly infinite set of relational tables ([2]). Intuitively, the meaning of the *rep* function is that given a c-table T, the function returns all relational tables that T represents under different valuations. In [2], this function is defined relative to the *open world assumption*. We define it relative to the closed world assumption. In the definition that follows, $main$, lc and gc are used to denote the main part, the local condition part, and the global condition part of a linear c-table, respectively. The symbol ε is used to denote the empty set.

Definition 2 (semantics of a linear c-table): A linear c-table T represents the set of relational tables that are defined by the following equation.

$$rep(T) = \{T' | \exists v, \text{ such that } \ v(T) = T'\} \qquad (1)$$

In the definition, v is a mapping that maps the variables in T to constants in the corresponding domains and is generalized to linear c-tuples as follows.

$$v(t) = \begin{cases} v(main(t)) & : \ v(lc(t)) \wedge v(gc(T)) \\ \varepsilon & : \ otherwise \end{cases} \qquad (2)$$

The value of $v(main(t))$ is calculated by substituting the variables in the main part of t with the values to which v maps them. The mapping v is further extended to linear c-tables as shown in Equation 3, where $\{t_i\}_{i=1}^{k}$ are the linear c-tuples in T.

$$v(T) = \{| \ v(t_i) | i \in [1, k] \wedge v(t_i) \neq \varepsilon \ |\} \qquad (3)$$

In the above definition, we have used the common notation $\{| \cdot |\}$ to denote a bag of elements. While it is possible to define an ordering on the linear c-tuples inside a linear c-table, we leave this topic as area for future research.

Definition 2 is novel and differs from the definitions presented in [2], [4]. Unlike these papers, we define duplicate semantics for c-tables and use the closed world assumption.

From now on, when the distinction is clear from the context, we will refer to linear c-tuples simply as c-tuples and to linear c-tables simply as c-tables.

III. SIMPLIFYING LINEAR C-TABLES

An important part of simplifying a linear c-table is simplifying the local conditions and the global condition, which are both expressed as linear conditions, and checking for their satisfiability. Details on how to simplify a linear condition and how to check if it is satisfiable under at least one valuation are presented next.

A. Linear Condition Simplification and Satisfiability Check

A *linear condition* is a Boolean expression and, as such, can be expressed as a disjunction of *positive conjunctions*. A positive conjunction is a conjunction of positive atomic linear conditions, where the later has the form $\bar{a} \cdot \bar{x} = \bar{b}$ or $\bar{a} \cdot \bar{x} < \bar{b}$ ($\bar{x}$ is a variable vector and $\bar{a}$ and $\bar{b}$ are vector constants). An *atomic linear condition* includes in addition negative conditions of the form $\bar{a} \cdot \bar{x} \neq \bar{b}$. An intuitive representation of a positive conjunction is a multi-dimensional polyhedron, which defines a semilinear set. Therefore, a linear condition can be interpreted as a set of disjoint polyhedra. Note however that, as shown if Figure 1, such a representation is not unique.

Let us first consider the algorithm that was proposed in [18] for normalizing conjunctions of linear equalities and inequalities. More precisely, the paper represents a conjunction of atomic linear conditions by the system $A\bar{x} \leq \bar{b}$, $E\bar{x} = \bar{d}$, $\neg(\bar{c}_i\bar{x} = \bar{f}_i)$, where A and E are matrices with constants, $\bar{b}$, $\bar{d}$, $\bar{c}_i$ and $\bar{f}_i$ are vectors of constants and $\bar{x}$ is a variable vector. The normalization algorithm

runs in polynomial time and relies on calls to a module that solves linear programs. We will refer to this algorithm as *normalize*.The algorithm has the added advantage that it recognizes sets of unsatisfiable atomic conditions and reports them as such by returning the empty set. Part of the algorithm deals with the elimination of redundant conditions, which is an extension of the research that is published in [19]. The pivot theorem from [18] follows.

Theorem 1: If two sets of atomic conditions over $\langle \mathbb{R}, \{+, >, =\} \rangle$ define the same point set, where $\mathbb{R}$ is the set of real numbers, then their canonical forms will have identical set of equality conditions, the same inequality conditions up to multiplication by a positive scalar, and the same set of negative conditions.

Algorithm 1 *simplify*(C)

1: $c_1 \vee c_2 \vee ... \vee c_n \leftarrow C$, where $\{c_i\}_{i=1}^{n}$ are positive conjunctions.
2: *result* $\leftarrow$ *break_up*$(c_1, \ldots, c_n)$
3: **if** *result* $= \emptyset$ **then**
4: **return false**
5: **end if**
6: **return** $g_1 \vee \cdots \vee g_m$, where $\{g_i\}_{i=1}^{m}$ are the conjunctions in *result*.

Algorithm 2 *break_up*$(c_1, \ldots, c_n)$

1: *result* $\leftarrow \{$*normalize*$(c_1)\}$
2: **for** $i \leftarrow 2$ to n **do**
3: **for** $g \in$ *result* **do**
4: *result* $\leftarrow$ *result* $\cup \{$*normalize*$(g \wedge c_i)\}$
5: *result* $\leftarrow$ *result* $\cup \{$*normalize*$(g \wedge \neg c_i)\}$
6: *result* $\leftarrow$ *result* $\cup \{$*normalize*$(\neg g \wedge c_i)\}$
7: *result* $\leftarrow$ *result* $- \{g\}$
8: **end for**
9: **end for**
10: **for** $g \in$ *result* **do**
11: **if** $g =$ **false then**
12: *result* $\leftarrow$ *result* $- \{g\}$
13: **end if**
14: **end for**
15: **return** *result*.

The pseudo-code for simplifying a linear condition C is presented in Algorithm 1. The algorithm first breaks C into a disjunction of positive conjunctions. Next, the algorithm divides the conjunctions so that they do not overlap. As a final step, the algorithm normalizes the conjunctions that are computed using the normalization algorithm from [18]. The following theorem address the correctness of the algorithm.

Theorem 2: Algorithm 1 is correct, that is, $C =$ *simplify*(C) for any linear condition C.

Proof: Line 1 of Algorithm 1 breaks C into disjunctive normal form and therefore does not change the value of C. Algorithm 2 breaks up conjunctions so that they do not overlap. The conjunction of the expressions $g \wedge c_i$, $g \wedge \neg c_i$, and $\neg g \wedge c_i$ is equal to $g \vee c_i$. Therefore, Lines 4-7 of the Algorithm 2 remove g from the set of conjunctions stored in *r*esult and add $g \vee c_i$. Therefore, the net effect of the lines is to add c_i to *result*. Therefore, after Lines 1-9 of Algorithm 2 the conjunctions $\{c_i\}_{i=1}^{n}$ are added to *result*. Lines 10-14 of Algorithm 2 remove `false` conjunctions from *result*. If after this process *result* is empty, then C is not satisfiable and `false` is returned correctly at Lines 4 of Algorithm 1. Line 6 of Algorithm 1 returns the computed disjoint conjunctions. ■

Theorem 3: Algorithm 1 runs in $O(m^c \cdot 3^n)$ time, where m is the length of the linear condition C, n is the number of conjunctions in the disjunctive normal form of C (i.e., $n \leq (\sqrt{2})^m$), and c is a constant.

Proof: In order to verify the running time of the algorithm, note that Line 1 takes $O(m \cdot n)$ time. Line 1 of Algorithm 2 makes a call to the normalization procedure from [18], which runs in $O(m^c)$ time (the length of each conjunction is smaller then the length of C). Lines 2-9 of Algorithm 2 make at most $\frac{3^n-1}{2} - 1$ calls to the procedure from [18]. The reason is that during the k^{th} iteration of the outer `for`-loop there can be as many as $3^{(k-2)}$ conjunctions in *result* and therefore as much as $3^{(k-1)}$ calls to the normalization procedure from [18]. Lines 10-14 of Algorithm 2 take less time to execute than Lines 2-9 of Algorithm 2 and therefore do not contribute to the complexity. Therefore, $\sum_{k=2}^{n} 3^{k-1} = \frac{3^n-1}{2} - 1$ is an upper bound on the number of calls to the normalization procedure and each call takes $O(m^c)$ time. ■

Algorithm 1 can be used to test the satisfiability of a linear condition. However, the part of the algorithm that removes the overlapping part of the conjunctions will no longer be needed. The modified pseudo-code is shown in Algorithm 3. We will refer to the simplified algorithm as *fast_simplify*. Alternative methods for testing for linear condition satisfiability are described in [20], [21]. The full-blown algorithm is only useful when we want to eliminate including the same point set multiple times by breaking up a linear condition into disjoint polyhedra.

Algorithm 3 *fast_simplify*(C)

1: $c_1 \vee c_2 \vee ... \vee c_n \leftarrow C$, where $\{c_i\}_{i=1}^{n}$ are positive conjunctions.
2: **for** $i \leftarrow 1$ to n **do**
3: $c_i \leftarrow$ *normalize*(c_i)
4: **end for**
5: **if** $c_i =$ **false** for $i = 1$ to n **then**
6: **return false**
7: **end if**
8: **return** $c_1 \vee \cdots \vee c_n$

Theorem 4: Algorithm 3 is correct, that is $fast\_simplify(C) = C$. Moreover, $fast\_simplify(C)$ returns `false` exactly when C is not satisfiable.

Proof: The algorithm breaks C into disjunctive normal form and normalizes each conjunction. This will not affect the value of C. Note that when C is not satisfiable each conjunction c_i will be evaluated as `false` (see [18] for a formal proof) and therefore the method will return `false`. ■

Theorem 5: The running time of Algorithm 3 is $O(n \cdot m^c) = O((\sqrt{2})^m \cdot m^c)$, where m is the length of the linear condition C, n is the number of conjunctions in the disjunctive normal form of C (i.e., $n \leq (\sqrt{2})^m$), and c is a constant.

Proof: Line 3 is executed n times and the complexity of the *normalize* method is $O(m^c)$. ■

The algorithm can be applied not only to conditions over the system $\langle \mathbb{R}, \{>, =, +\}\rangle$, but also to conditions over the system $\langle \mathbb{R}, \{>, =, +\}\rangle \cup \langle \mathbf{S}, \{=, \neq\}\rangle$. To do so, substitute each atomic conditions of the form $x \neq c$, where x is a string variable and c is a string constant with $x = c_1 \vee x = c_2 \vee \cdots \vee x = c_r \vee x = c_{r+1}$, where c_{r+1} is a newly introduced string constant and $\{c_i\}_{i=1}^{r}$ are the existing string constants excluding c. In other words, we pin the value of x to be equal to one of the existing constants (excluding c) or to a new constant, which is equivalent to stating that $x \neq c$.

Similarly, substitute each atomic condition of the form $x \neq y$, where x and y are string variables with $\bigvee_{i,j=1,r+2}^{i \neq j} (x = c_i \wedge y = c_j)$, where c_{r+1} and c_{r+2} are newly introduced constants and $\{c_i\}_{i=1}^{r}$ are the existing string constants. In other words, we add the restriction on the variables x and y that they are equal to distinct constants, which implies $x \neq y$.

Alternatively, Line 1 of the algorithm can be modified to require the breaking of C into not necessarily positive conjunctions. This modification allows the direct application of the normalization algorithm to a linear condition containing strings because the algorithm from [18] handles inequality conditions in addition to equality and weak-inequality (i.e., greater than and less than) conditions.

B. C-Table Simplification

Note that there may be different c-tables representing the same set of bag relational tables, that is, it may be the case that $T_1 \neq T_2$ but $rep(T_1) = rep(T_2)$. The following definition formally defines the concept of c-table equivalence.

Definition 3 (c-table equivalence): If $rep(T_1) = rep(T_2)$, then we will say that T_1 and T_2 are equivalent and write $T_1 \approx T_2$.

Algorithm 4 shows how to simplify a c-table. The algorithm relies on the notion of c-tuple unification, which is defined next.

A	*B*	*condition*
1	2	$x = 1$
z	2	$x = 2$
p	w	$x = t$

g.c.: $t \neq 1 \wedge t \neq 2$

A	*B*	*condition*
a	b	$((a = 1) \wedge (b = 2) \wedge (x = 1)) \vee$ $((a = z) \wedge (b = 2) \wedge (x = 2)) \vee$ $((a = p) \wedge (b = w) \wedge (x = t))$

g.c.: $t \neq 1 \wedge t \neq 2$

Table II
A C-TABLE AND THE RESULT OF APPLYING STEPS 1 AND 2 OF THE C-TABLE SIMPLIFICATION ALGORITHM

Definition 4 (c-tuple unification): The c-tuples t_1 and t_2 of the c-table T are unifiable exactly when the formula $lc(t_1) \wedge lc(t_2) \wedge gc(T)$ is not satisfiable. We will denote this check as *unifiable*, that is $unifiable(t_1, t_2) = \neg(lc(t_1) \wedge lc(t_2) \wedge gc(T))$.

Algorithm 4 *simplify*(T)

1: **for** $t \in T$ **do**
2: **if** *fast_simplify*($lc(t) \wedge gc(t)$) = **false then**
3: remove t from T
4: **end if**
5: **end for**
6: **while** $\exists\{t_1, t_2\}$ s.t. *unifiable*(t_1, t_2) **do**
7: remove t_1 and t_2 from T
8: create c-tuple t with main part $\bar{X} = x_1, x_2, ..., x_n$, where n is the arity of T and $\{x_i\}_{i=1}^{n}$ are newly introduced variables.
9: add to t the local condition $(\bar{X} = main(t_1) \wedge lc(t_1)) \vee (\bar{X} = main(t_2) \wedge lc(t_2))$
10: add t to T
11: **end while**
12: **for** $t \in T$ **do**
13: $lc(t) \leftarrow$ *simplify*($lc(t) \wedge gc(T)$)
14: **if** $lc(t) =$ **false then**
15: remove t from T
16: **else**
17: **while** $main(t)$ contains the variable x for an attribute and *fast_simplify*($lc(t) \Rightarrow (x = c)$) = **true do**
18: replace x with constant c in $main(t)$
19: **end while**
20: **end if**
21: **end for**
22: $gc(T) \leftarrow$ **true**
23: **return** T

The intuition behind the definition is that if two c-tuples have local conditions that cannot both hold under any valuation, then at most one of the c-tuples could be present in any representation of the c-table and therefore the two

c-tuples can be merged into a single c-tuple.

Table II shows the result of applying the first eleven lines of the algorithm. The following theorem captures the correctness of Algorithm 4.

Theorem 6: Algorithm 4 is correct, that is, $simplify(T) \approx T$ for any c-table T.

Proof: Lines 1-5 of the algorithm remove c-tuples that are not part of any representation. Therefore, they will not have an effect on *rep*(T). Lines 6-11 of the algorithm unify c-tuples that can be unified and thus reducing the size of the c-table without changing its representations. The reason is that two c-tuples that have incompatible local conditions cannot both appear in any representation. Lines 12-22 of of the algorithm move the global condition to the local conditions and simplify the resulting local conditions. Again, this will not affect the set of representations for the table. ■

The next theorem describes the time complexity of Algorithm 4.

Theorem 7: Algorithm 4 runs in $O(d^3 \cdot (\sqrt{2})^{d \cdot m} \cdot (d \cdot m)^c + (m \cdot d)^c \cdot 3^{(\sqrt{2})^{m \cdot d}} \cdot n)$ time, where n is the number of c-tuples, m is the greater of the size of the longest c-tuple and the size of the global condition, d is the number of c-tuples with non-trivial local conditions (i.e., local conditions that are different than `true`), and c is a constant.

Proof: Lines 1-5 of the algorithm takes $O((\sqrt{2})^m \cdot m^c \cdot d)$ time. The reason is that algorithm *fast_simplify*, which takes $O((\sqrt{2})^m \cdot m^c)$ time, needs to be applied to d local conditions.

Lines 6-11 will take $O(d^3 \cdot (\sqrt{2})^{d \cdot m} \cdot (d \cdot m)^c)$ time. The reason is that $\binom{d}{2} + \binom{d-1}{2} + \cdots + \binom{2}{2} = O(d^3)$ iterations of the while loops can be performed and each iteration can take as much as $O((\sqrt{2})^{d \cdot m} \cdot (d \cdot m)^c)$ time because we use the *fast_simplify* algorithm on expressions as long as $d \cdot m$ when checking if two c-tuples are unifiable.

Line 13, which has the highest time complexity in the loop defined by Lines 12-21, can be applied on a local condition as big as $m \cdot d$ and therefore takes $O((m \cdot d)^c \cdot 3^{(\sqrt{2})^{m \cdot d}})$ time. The line can be applied at most n times.

Line 22 can be be executed in constant time. ■

Note that Algorithm 4 can be improved by applying dynamic program or iterative dynamic programming techniques ([22]). For example, we can buffer existing results and use them in performing new calculations. This approach will save time because most of the presented algorithms produce c-tuples with local conditions that have subexpressions in common.

Note as well that Algorithm 4 does not produce a canonical form for c-tables. In order to understand why it is challenging to create a canonical form for c-tables, consider the first c-table from Table III. As the table shows, there are two different ways to apply Algorithm 4. Since the algorithm is non-deterministic, applying the algorithm differently yields different results. Therefore, the purpose of Algorithm 4 is not to find a canonical form for c-tables but to simplify a c-table. Since, as Table III suggest, unifying two c-tuples can prevent us from unifying one of the c-tuple with a third c-tuple, the problem of finding a canonical form for c-tuples is intrinsically hard.

A	*condition*
1	$x < 2$
1	$3 < x < 5$
1	$4 < x < 6$

A	*condition*
1	$x < 2 \vee 3 < x < 5$
1	$4 < x < 6$

A	*condition*
1	$x < 2 \vee 4 < x < 6$
1	$3 < x < 5$

Table III
AN EXAMPLE C-TABLE SIMPLIFICATION

C. Checking for C-Table Equality

As we have seen so far, c-tables are different from relational tables because they allow multiple ways to represent the same information. Therefore, checking for c-table equality is not trivial. The following theorem describes one possible way to do so.

Theorem 8: Two c-tables T_1 and T_2 represent the same set of tables exactly when *simplify*$(T_1 \dot{-} T_2) = \emptyset$ and *simplify*$(T_2 \dot{-} T_1) = \emptyset$, where "$\dot{-}$" is the monus operation that is introduced in the next section.

Proof: $\Rightarrow$ Let T_1 and T_2 represent the same set of tables. Then *simplify*$(T_1 \dot{-} T_2) \approx \emptyset$ and *simplify*$(T_2 \dot{-} T_1) \approx \emptyset$. However, note that if a c-table represents the empty set, then all local conditions will be unsatisfiable after Line 13 of Algorithm 4 and Lines 14-15 will remove all c-tuples from the c-table and make it empty. Therefore, it will be the case that *simplify*$(T_1 \dot{-} T_2) = \emptyset$ and *simplify*$(T_2 \dot{-} T_1) = \emptyset$.

$\Leftarrow$ Let *simplify*$(T_1 \dot{-} T_2) = \emptyset$ and *simplify*$(T_2 \dot{-} T_1) = \emptyset$. Then $rep(T_1 \dot{-} T_2) = \emptyset$ and $rep(T_2 \dot{-} T_1) = \emptyset$ and therefore it must be the case that T_1 and T_2 represent the same set of relational tables. ■

IV. BAG RELATION ALGEBRA FOR C-TABLES

So far, we have defined the syntax and semantics of c-tables and presented an algorithm for their simplification. Next, we will describe how relational algebra[2] can be extended to handle c-tables. Specifically, since we are using the closed world assumption, we are able to develop a *sound* and *complete* extension of relational algebra that is *closed*. The definition of three terms follows.

Definition 5 (closed relational algebra): A relational algebra is *closed* exactly when the result of applying any operator q with arity n of the relational algebra to the c-tables $\{T\}_{i=1}^{n}$ produces a c-table, that is, $q(T_1, \ldots, T_n)$ is always a c-table.

[2]Our choice of relational algebra is arbitrary, that is, any language with the expressive power of relational algebra, such as relational calculus, can be used instead.

Definition 6 (sound relational algebra): A relational algebra is *sound* exactly when only correct answers appear in the result of $q(T_1, \ldots, T_n)$ or formally $rep(q(T_1, \ldots, T_n)) \subseteq q(rep(T_1, \ldots, T_n))$ for any c-tables $\{T_i\}_{i=1}^n$ and operator q with arity n.

Note that throughout the paper we use $q(rep(T_1, \ldots, T_n))$ to denote the result of applying q to each table in the set $rep(T_1, \ldots, T_n)$.

Definition 7 (complete relational algebra): A relational algebra is *complete* exactly when all correct answers appear in the result of $q(T_1, \ldots, T_n)$ or formally $q(rep(T_1, \ldots, T_n)) \subseteq rep(q(T_1, \ldots, T_n))$ for any c-tables $\{T_i\}_{i=1}^n$ and operator q with arity n.

A relational algebra operator is *well defined* exactly when it is closed, sound, and complete. In this section we define the semantics of *projection*, *selection*, *inner join*, *union*, *monus*, and *duplicate elimination* over c-tables with bag semantics and show that all operators are well defined. The grouping and aggregation operations are discussed in the next section.

A. Projection

Definition 8 (syntax and semantics of projection): If T is a c-table with attributes $\bar{A}$, then we denote the projection of the attributes $\bar{A}'$ over this c-table as $\pi_{\bar{A}'}(T)$. The pseudo-code for performing the projection operator is shown in Algorithm 5.

The c-table $\pi_{\bar{A}'}(T)$ is constructed from the c-table T by removing all columns in $\bar{A} - \bar{A}'$ and leaving the same local and global conditions.

Algorithm 5 $\pi_{\bar{A}'}(T)$

```
for t ∈ T do
    remove attributes outside the set A from t
end for
return T
```

Note that the above definition defines duplicate-preserving projection. The duplicate-eliminating projection, which is more common in the relational model, can be constructed by applying the duplicate-elimination operator to the result of applying the duplicate-preserving projection. Algorithm 5 does not remove conditions that include variables associated with removed attributes because these conditions are still relevant. For example, even if an attribute that contains the variable x is removed from Table I, the variable x should not be removed from the local conditions because it stores the information that only one of the first two c-tuples can appear in any representation.

Theorem 9: The projection operator is well defined.

Proof: We need to show that $rep(\pi_{\bar{A}}(T)) = \pi_{\bar{A}}(rep(T))$.
$\Rightarrow$ Let $T_1 \in rep(\pi_{\bar{A}}(T))$, where T_1 is a relational table. Then there exists a valuation v such that $T_1 = v(\pi_{\bar{A}}(T))$. Let T_2 be a relational table that extends T_1 with arbitrary values for the attributes outside the set $\bar{A}$. Then the equation $T_1 = \pi_{\bar{A}}(T_2)$ will hold. Let v' be the valuation v extended so that $T_2 = v'(T)$. Then $T_2 \in rep(T)$ and therefore $T_1 \in \pi_{\bar{A}}(rep(T))$.

$\Leftarrow$ Let $T_1 \in \pi_{\bar{A}}(rep(T))$. Then there exists valuation v such that $T_1 = \pi_{\bar{A}}(v(T))$. Let T_2 be a relational table that extends T_1 with arbitrary values for the attributes outside the set $\bar{A}$. Then the equation $T_1 = \pi_{\bar{A}}(T_2)$ will hold. Note that $T_1 = v'(\pi_{\bar{A}}(T))$ where v' is a valuation that extends v to the attributes outside the set $\bar{A}$ according to the values of the attributes in T_2. Therefore $T_1 \in rep(\pi_{\bar{A}}(T))$. ■

A	B	*condition*
2	x	$x \neq 3$
2	4	TRUE

g.c. $x \neq 2$

B	C	*condition*
4	1	TRUE
2	z	$z > 3$

g.c. TRUE

Table IV
EXAMPLE R_1 AND R_2 C-TABLES

B	*condition*
4	TRUE
2	$z > 3$

g.c. TRUE

B	C	*condition*
4	1	TRUE $\wedge$ $1 > 2$
2	z	$z > 3 \wedge z > 2$

g.c. TRUE

Table V
THE RESULT OF $\pi_B(R_2)$ AND $\sigma_{C>2}(R_2)$

Table IV shows two example c-tables that we will use throughout this section. The left part of Table V shows the result of $\pi_B(R_2)$. The following theorem describes the complexity of the projection operator.

Theorem 10: The projection operator takes $O(s)$ time, where s is the size of the c-table on which the projection is applied.

Proof: The operator goes though the c-tuples of the c-table exactly once and eliminates certain attributes. Therefore, the time complexity of the operator is equal to order the size of the c-table. ■

B. Selection

Definition 9 (syntax and semantics of selection): We denote the selection over a c-table T as $\sigma_\gamma(T)$, where γ is a predicate formula over $\langle \mathbb{R}, \{>, =, +\} \rangle \cup \langle \mathbf{S}, \{=, \neq\} \rangle$ that references the variables $\{A_i\}_{i=1}^n$ that have the same names as the attributes of T. The pseudo-code for performing selection is presented in Algorithm 6.

Algorithm 6 $\sigma_\gamma(T)$

```
for t ∈ T do
    θ(t) ← a substitution that substitutes every variable
           A_i with t[A_i] (the value for the attribute A_i in t).
    lc(t) ← lc(t) ∧ γ_θ(t)
end for
return T
```

Theorem 11: The selection operator is well defined.

Proof: We need to show that $rep(\sigma_\gamma(T)) \equiv \sigma_\gamma(rep(T))$ for every c-table T. But this is equivalent to proving that there exists valuations v and v' s.t. $v(\sigma_\gamma(T)) = \sigma_\gamma(v'(T))$. However, we have defined selection over c-tables in such a way so that $v(\sigma_\gamma(T)) = \sigma_\gamma(v(T))$ for any valuation v, which proves that selection is well defined. ■

The right part of Table V shows the result of $\sigma_{C>2}(R_2)$. The following theorem proves the time complexity of the selection operator.

Theorem 12: The selection, as we have defined it, takes $O(s * m)$ time, where s is the size of the c-table and m is the size of the selection condition.

Proof: The number of c-tuples in the c-table is bounded by s. For every c-tuple, we need to add to its local condition a condition of size m and therefore the time complexity of the algorithm is $O(s * m)$. ■.

In order to save space, c-tuples with unsatisfiable local condition can be removed from the c-table, where we can use the *fast_simplify* algorithm to detect such c-tuples.

C. Inner Join

Definition 10 (syntax and semantics of inner join): Consider a c-table T_1 with attributes $\{\bar{A}, \bar{B}\}$ and a c-table T_2 with attributes $\{\bar{B}, \bar{C}\}$. We denote the inner join of T_1 and T_2 on the set of attributes $\bar{B}$ as $T_1 \bowtie_{\bar{B}} T_2$. The pseudo-code for performing inner join is presented in Algorithm 7.

Algorithm 7 $T_1 \bowtie_{\bar{B}} T_2$

```
1: T ← empty c-table with attributes Ā ∪ B̄ ∪ C̄
2: rename the variables in T2 so that T1 and T2 do no share
   variables
3: for t1 ∈ T1 do
4:    for t2 ∈ T2 do
5:       if fast_simplify(π_B̄(main(t1)) =
         π_B̄(main(t2)))! = false then
6:          main(t) ← (t1, π_C̄(t2))
7:          lc(t) ← lc(t1) ∧ lc(t2) ∧ (t1[B̄] = t2[B̄])
8:          add t to T
9:       end if
10:   end for
11: end for
12: gc(T) ← gc(T1) ∧ gc(T2)
13: return T
```

Theorem 13: The inner join operator is well defined.

Proof: Let v be a valuation of the distinct variables of T_1 and T_2 (after Line 2 of Algorithm 7 is executed). We need to show that $v(T_1 \bowtie_{\bar{B}} T_2) = v(T_1) \bowtie_{\bar{B}} v(T_2)$. Let $t \in v(T_1 \bowtie_{\bar{B}} T_2)$. Then there must exist $t_1 \in T_1$ and $t_2 \in T_2$ such that the main part of t is equal to the join of the main parts of t_1 and t_2. Then $t = v(t_1) \bowtie_{\bar{B}} v(t_2)$ and therefore $t \in v(T_1) \bowtie_{\bar{B}} v(T_2)$. The other direction is analogous. ■

A	*B*	*C*	*condition*
2	x	1	$x \neq 3 \wedge x = 4 \wedge$ TRUE
2	x	z	$x \neq 3 \wedge x = 2 \wedge z > 3$
2	4	1	TRUE $\wedge$ TRUE

g.c. $x \neq 2 \wedge$ TRUE

Table VI
THE RESULT OF $R_1 \bowtie R_2$

Table VI shows the result of $R_1 \bowtie R_2$. The following theorem proves the time complexity of the inner join operator.

Theorem 14: Inner join takes $O(n' \cdot n'' \cdot (\sqrt{2})^m \cdot m^c)$ time, where n' and n'' are the sizes of the c-tables that are being joined, m is the size of the longest local condition in them, and c is a constant.

Proof: The code inside the double `for`-loop is executed $n' \cdot n''$ number of times. The main time complexity in Lines 5-9 come from the call to the *fast_simplify* method, which takes $O((\sqrt{2})^m \cdot m^c)$ time to execute. ■.

D. Union

Definition 11 (syntax and semantics of union): If T_1 and T_2 are c-tables, then we will denote their union as $T_1 \cup T_2$. The pseudo-code for calculating the union of c-tables is presented in Algorithm 8.

Algorithm 8 $T_1 \cup T_2$

```
1: T ← empty c-table
2: rename the variables in T2 so that T1 and T2 do no share
   variables
3: for t1 ∈ T1 do
4:    add t1 to T
5: end for
6: for t2 ∈ T2 do
7:    add t2 to T
8: end for
9: gc(T) ← gc(T1) ∧ gc(T2)
10: return T
```

Note that we define the union operator to be duplicate preserving. Duplicate eliminating union can be performed by applying duplicate elimination to its result.

Theorem 15: The union operator is well defined.

Proof: We need to show that $v(T_1 \cup T_2) = v(T_1) \cup v(T_2)$ for any valuation v. Let $t \in v(T_1 \cup T_2)$. Then there exit c-tuple t_1 such that t_1 is in either T_1 or T_2 and $t = v(t_1)$. Therefore, $t \in v(T_1) \cup v(T_2)$. The reverse direction is analogous. ■.

The following theorem proves the time complexity of Algorithm 8.

Theorem 16: The time complexity of Algorithm 8 is $O(n + m)$ where n and m are the sizes of T_1 and T_2, respectively.

Proof: The time complexity of the algorithm comes from Lines 3-5, which take $O(n)$ time, and Lines 6-8, which take take $O(m)$ time. Therefore, the total time complexity of the algorithm is $O(n+m)$. ■.

E. Monus

In bag relational algebra over bag relational tables monus is defined as: $T_1 \dot{-} T_2 = \{t_{[k]} | t \in T_1 \wedge k = max(count(t, T_1) - count(t, T_2), 0)\}$, where $t_{[k]}$ is used to denote the tuple t replicated k times and $count$ is a function that returns the number of occurrences of the tuple specified as the first parameter in the table specified as the second parameter. The following definition extends the monus operator to c-tables.

Definition 12 (syntax and semantics of monus): The monus of two c-tables T_1 and T_2 is defined as $T_1 \dot{-} T_2$. The pseudo-code for performing the monus operator is presented in Algorithm 9.

Algorithm 9 $T_1 \dot{-} T_2$

1: rename the variables in T_2 so that T_1 and T_2 do no share variables
2: $V \leftarrow T_1$
3: $i \leftarrow 0$
4: **for** $t_1 \in T_1$ **do**
5: $\quad j \leftarrow 0$
6: $\quad$ **for** $t_2 \in T_2$ **do**
7: $\quad\quad X[i][j] = (main(t_1) = main(t_2)) \wedge lc(t_1) \wedge gc(T_1) \wedge lc(t_2) \wedge gc(T_2)$
8: $\quad\quad j \leftarrow j + 1$
9: $\quad$ **end for**
10: $\quad i \leftarrow i + 1$
11: **end for**
12: $gc(V) \leftarrow gc(V) \wedge \bigwedge_{j=1}^{m} [\bigvee_{i=1}^{n} (Y[1,j] = \cdots = Y[i-1,j] = Y[i+1,j] = \cdots = Y[n,j] = 0 \wedge Y[i,j] = 1)] \wedge \bigwedge_{i=1}^{n} [\bigvee_{j=1}^{m} (Y[i,1] = \cdots = Y[i,j-1] = Y[i,j+1] = \cdots = Y[i,m] = 0 \wedge Y[i,j] = 1)]$
13: **for** $t \in V$ **do**
14: $\quad lc(t) \leftarrow lc(t) \wedge \neg[\bigvee_{j=1}^{m} (X[i,j] \wedge (Y[i,j] = 1))]$
15: **end for**
16: **return** V

Theorem 17: The monus operator is well defined, where the definition of complete is changed to: $[Rep(T') \dot{-} Rep(T'')] \cup \{\emptyset\} \subseteq Rep(T' \dot{-} T'')$.

Proof: The algorithm first renames the variables of T_2 so that they are distinct from those in T_1. Next, it calculates the matrix X and sets a restriction on the possible values for the matrix Y. The value of $X[i,j]$ contains the condition that must hold for the i^{th} c-tuple of T_1 to be deleted from T_1 and the c-tuple that "deletes" it to be the j^{th} c-tuple of T_2. The matrix $Y[i][j]$ has the restriction that for each j there exists exactly one i such that Y[i][j]=1 and that for each i there exists exactly one j such that $Y[i][j] = 1$ (the elements of the matrix Y can only take the values 0 and 1). The matrix Y is used to enforce the condition that every c-tuple t_2 in T_2 can be used to delete at most one c-tuple of T_1 and that every c-tuple t_1 in T_2 can be deleted at most once. Lastly, the local conditions that we add to the resulting c-table do the deletions. They specify that if for some valuation both $X[i][j]$ and $(Y[i][j] = 1)$ hold, (i.e., if a c-tuple t'_i in T' matches with a c-tuple t''_j in T'' and the valuation is such that t'_i can not be deleted by any c-tuple other then t'_j and t'_j can only delete t'_i), then the c-tuple that was constructed from the i^{th} c-tuple in T_1 should be deleted from the resulting c-table V.

Given a valuation v, each c-tuple in T_1 will be deleted only if there exists a matching c-tuple in T''. Moreover, given a valuation v, every c-tuple in T_2 can delete at most one c-tuple from T_1. Therefore, the algorithm is correct and $Rep(T_1 \dot{-} T_2) \equiv [Rep(T_1) \dot{-} Rep(T_2)] \cup \{\emptyset\}$. Here $\{\emptyset\}$ is used to represent the empty c-table. ■

Note that we had to modify the definition of a complete relational algebra because we constructed the global condition of $T_1 \dot{-} T_2$ in such a way so that we allow for $\{\emptyset\}$ to be a possible representation. It is our believe that this is an intrinsic problem of monus when dealing with the closed world assumption.

A demonstration of how monus can be applied over the example c-tables from Table IV is shown in Table VII. The following theorem describes the time complexity of Algorithm 9.

$x \neq 3 \wedge x \neq 2 \wedge$ $x = 4 \wedge$ TRUE $\wedge$ TRUE	$x \neq 3 \wedge x \neq 2 \wedge$ $x = 2 \wedge z > 3 \wedge$ TRUE
TRUE $\wedge\ x \neq 2 \wedge 4 = 4$ TRUE $\wedge$ TRUE	FALSE

A	*B*	*condition*
2	x	$x \neq 3 \wedge \neg((X[1,1] \wedge$ $Y[1,1] = 1) \vee (X[1,2] \wedge Y[1,2] = 1))$
2	4	TRUE $\wedge\ \neg((X[2,1] \wedge$ $Y[2,1] = 1) \vee (X[2,2] \wedge Y[2,2] = 1))$

g.c. $(x \neq 2) \wedge ((Y[1,1] = Y[2,2] = 1 \wedge Y[1,2] = Y[2,1] = 0) \vee (Y[1,2] = Y[2,1] = 1 \wedge Y[1,1] = Y[2,2] = 0))$

Table VII
SHOWS THE MATRIX X AND THE RESULT FOR $R_1 \dot{-} R_2$

Theorem 18: Monus, takes $O(m \cdot n)$ time, where m and m are the sizes of the c-tables on which the operation is performed.

Proof: The complexity of the algorithm comes from the two `for`-loops. The code inside the double `for`-loops runs in constant time and it is executed $O(m \cdot n)$ time. ■

A	B	*condition*
2	x	$x \neq 3 \wedge (x \neq 4 \vee$ FALSE)
2	4	TRUE $\wedge (x \neq 4 \vee x = 3)$

g.c. $x \neq 2$

Table VIII
THE RESULT OF $\varepsilon(R_1)$

F. Duplicate Elimination

The last relational algebra operation that we will explore in this section is duplicate elimination. In the relational case, duplicate elimination can be defined as a grouping on all the attributes. We adopt similar definition here.

Definition 13: We will denote the duplicate elimination operator applied to the c-table T as $\varepsilon(T)$. We will compute $\varepsilon(T)$ using the formula $\varepsilon(T) = group_{\bar{A}}(T)$, where $\bar{A}$ are the attributes of T.

Note that the result of the group operation is a *nested c-table* (see Table IX for an example of a nested c-table). We define the semantics of a nested c-table and of the *group* operation in Section V-A.

Theorem 19: The duplicate elimination operator is well defined.

Proof: $\varepsilon(Rep(T)) \equiv group_{\bar{A}}(Rep(T)) \equiv Rep(group_{\bar{A}}(T)) \equiv Rep(\varepsilon(T))$, which proves that duplicate elimination is well defined. The fact that the equation $group_{\bar{A}}(Rep(T)) \equiv Rep(group_{\bar{A}}(T))$ holds follows from the fact that the $group$ operation is well defined over c-tables, which will be proven in Section V-A. ■

The result of $\varepsilon(R_1)$, where R_1 is the c-table defined in Table IV, is shown in Table VIII.

V. APPLYING AGGREGATION TO C-TABLES

To the best of our knowledge, no research has been previously published in the area of applying grouping and aggregation to c-tables. We are aware of research on applying aggregation to fuzzy numbers ([23]) and to random variables ([24]), but the query results in these algorithms are approximations. On the other hand, the research done in constraint databases ([25]) has explored the problem of aggregation over constraint databases. Unfortunately, the operation of aggregation in most constraint database systems is not closed ([26]). We are also aware of recent research in the area of auditing confidential information ([27]), which, however, deals only with aggregation over Boolean variables.

In general, we would like to be able to evaluate a relational expression of the form ${}_{\bar{A}}\mathcal{F}_{agg_1(B_1),\ldots,agg_n(B_b)}T$, where $\bar{A} \cup \{B_i\}_{i=1}^{b}$ are the attributes of T, $\bar{A} = \{A_i\}_{i=1}^{a}$, and each agg_i is one of the aggregates: *min*, *max*, *sum*, *count* and *avg*. In the relational case, the above expression is evaluated by grouping the tuples that have the same value for the attributes $\bar{A}$ into a single tuple that has this common value for the attributes $\bar{A}$. The value for the remaining attributes is calculated by applying the aggregation operations $\{agg_i\}_{i=1}^{n}$ to the value of the $\bar{B}$ attributes of the tuples in the group. In order to extend this definition to c-tables, we will need to be able to group c-tuples and perform aggregation on c-tuples.

A. Grouping

The result of the grouping operation is a nested c-table that consists of nested c-tuples. An example nested c-table is shown in Table IX. Informally, a nested c-table consists of c-tuples that can have more than one value for some of the attributes. A formal definition follows.

Definition 14 (nested c-tables and c-tuples): A *nested c-tuple* with single valued attributes $\{A_i\}_{i=1}^{a}$ and multi-valued attributes $\{B_i\}_1^b$ is the sequence of mappings from A_i to $D(A_i) \cup V_i$ for i ranging from 1 to a plus the sequence of mapping from B_i to a bag of values over $D(B_i) \cup V_i$ for i ranging from 1 to b plus a local condition over $\langle \mathbb{R}, \{>, =, +\}\rangle \cup \langle \mathbf{S}, \{=, \neq\}\rangle$. Note that here $D(A)$ is used to denote the domain of A and V_i is used to represent a possibly infinite, but countable, set of variables over $D(A_i)$ in the first case and over $D(B_i)$ in the second case.

A *nested c-table* is a c-table that contains nested c-tuples. The semantics of a nested c-table is similiar to the semantics of a regular c-table as described in Section II-A (see Equations 1, 2, and 3). The only difference is that a nested c-table represents a set of nested bag relational tables (see [28]) under different valuations and consists of a bag of nested c-tuples.

The algorithm for performing the grouping uses the concept of a semi-unifiable c-tuples and the $\prec$ relation for c-tuples, which are formally presented next.

Definition 15 (semi-unifiable c-tuples): The c-tuples $\{t_i\}_{i=1}^{n}$ of the c-table T are semi-unifiable relative to the set of attributes $\bar{A}$ exactly when the expression $\bigwedge_{i,j=1}^{n} \pi_{\bar{A}}(main(t_i)) = \pi_{\bar{A}}(main(t_j))$ is satisfiable under some valuation.

Informally, a bag of c-tuples are semi-unifiable relative to the set of attributes $\bar{A}$ exactly when the c-tuples can be potentially grouped into a single nested c-tuple in which $\bar{A}$ are the single-valued attributes.

Definition 16 (the $\prec$ relation): We will write $t_1 \prec_A t_2$, where t_1 and t_2 are c-tuples and $\bar{A}$ is a set of attributes exactly when $main(\pi_{\bar{A}}(t_2))$ can be constructed from $main(\pi_{\bar{A}}(t_1))$ by substituting some of the variables in $main(\pi_{\bar{A}}(t_1))$ with constants.

Informally, the $\prec$ relation compares the main parts of two c-tuples to determine if one c-tuple has more specific values than the other. The $\prec$ relation is transitive and defines partial order.

Definition 17 (syntax and semantics of grouping): We denote the result of grouping by the attributes $\bar{A}$ of T as

A	B	*condition*
2	x	$x \neq 3 \wedge (x \neq 4 \vee \text{FALSE})$
2	4	TRUE $\wedge$ $(x \neq 4 \vee x = 3)$
2 2	4	$x \neq 3 \wedge$ TRUE $\wedge$ $x = 4$

g.c. $x \neq 2$

Table IX
THE RESULT OF $group_B R_1$

A	B	C	condition
x	y	1	$(x + y = 3) \vee (x > 4) \vee (x < 0)$
x	3	2	$(x + y = 3 \wedge x < 2) \vee (x < 0)$
2	3	3	$x > 5$
2	3	4	TRUE
3	4	5	TRUE

Table X
EXAMPLE C-TABLE R

$group_{\bar{A}}(T)$. The pseudo-code for performing the grouping operator is shown in Algorithm 10.

The result of the grouping operation will be a *nested c-table*, that is, the value of a field in it may be a bag of values. For example, in $group_{\bar{A}}(T)$ the values for the attributes in $\bar{A}$ will be single values and for the the rest of the attributes - bag of values. The result of $group_B R_1$ is shown in Table IX, where R_1 is shown in Table IV.

Theorem 20: The *group* operator is well defined, that is, $group_{\bar{A}}(Rep(T)) \equiv Rep(group_{\bar{A}}(T))$.

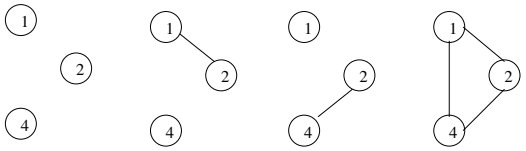
Proof: Line 1 copies T into the resulting c-table (our running example is on the the c-table R shown in Table XVIII and we show how to calculate $group_{A,B}(R)$).

Line 2 clusters the c-tuples into e-bags relative to the attributes of $\bar{A}$. Table XI shows the two e-bags that will

A	B	C	*condition*
x	y	1	$((x + y = 3) \vee (x > 4) \vee (x < 0)) \wedge t = 1$
x	3	2	$(x + y = 3 \wedge x < 2) \vee (x < 0)$
2	3	3	$x > 5$
2	3	4	TRUE

A	B	C	*condition*
x	y	1	$((x + y = 3) \vee (x > 4) \vee (x < 0)) \wedge t \neq 1$
3	4	5	TRUE

Table XI
E-BAGS IN $group_{A,B}(R)$



Graph 1 Graph 2 Graph 3 Graph 4

Figure 2. The four possible graphs for the first e-bag

$C[i]$	value	c-tuples
$C[1]$	$x < 0 \wedge t = 1$	$\{1, 2, 4\}$
$C[2]$	$0 \leq x < 2 \wedge x + y = 3 \wedge t = 1$	$\{1, 2, 4\}$
$C[3]$	$2 \leq x \leq 4 \wedge x + y = 3 \wedge t = 1$	$\{1, 4\}$
$C[4]$	$4 < x \leq 5 \wedge t = 1$	$\{1, 4\}$
$C[5]$	$x > 5 \wedge t = 1$	$\{1, 3, 4\}$
$C[6]$	$x < 0 \wedge t \neq 1$	$\{2, 4\}$
$C[7]$	$0 \leq x < 2 \wedge x + y = 3 \wedge t \neq 1$	$\{2, 4\}$
$C[8]$	$x > 5 \wedge t \neq 1$	$\{3, 4\}$
$C[9]$	$(x + y \neq 3 \wedge 0 \leq x \leq 4) \vee (4 < x \leq 5 \wedge t \neq 1)$	$\{4\}$

$C[i]$	value	c-tuples
$C[1]$	$((x + y = 3) \vee (x > 4) \vee (x < 0)) \wedge t \neq 1$	$\{1, 2\}$
$C[2]$	$((x + y \neq 3) \wedge (0 \leq x \leq x)) \vee (t = 1)$	$\{2\}$

Table XII
THE ARRAY C FOR THE TWO E-BAGS

$D[i]$	values	c-tuples
$D[1]$	$C[1] \vee C[2]$	$\{1, 2, 4\}$
$D[2]$	$C[3] \vee C[4]$	$\{1, 4\}$
$D[3]$	$C[5]$	$\{1, 3, 4\}$
$D[4]$	$C[6] \vee C[7]$	$\{2, 4\}$
$D[5]$	$C[8]$	$\{3, 4\}$
$D[6]$	$C[9]$	$\{4\}$

$D[i]$	values	c-tuples
$D[1]$	$C[1]$	$\{1, 2\}$
$D[2]$	$C[2]$	$\{2\}$

Table XIII
THE ARRAY D FOR THE TWO E-BAGS

be constructed after applying Line 2 to our example. Note that Line 2 is equivalence preserving and that c-tuples from different e-bags cannot contribute to the same resulting nested c-tuple under any valuation. This is why it suffices to perform the *group* operation to the c-tuples in each e-bag and then merge the results.

Line 3 partitions each e-bag further into r-bags. In other words, we partition the space over which the local conditions of the c-tuples in the e-bags is defined into non-overlapping polyhedra. Each r-bag corresponds to a set of disjoint

A	B	C	*condition*
x	y	1	$y \neq 3 \wedge x \neq 2 \wedge R$
x	3	2	$y \neq 3 \wedge x \neq 2 \wedge R$
2	3	4	$y \neq 3 \wedge x \neq 2 \wedge R$
x	y	1 2	$x \neq 2 \wedge y = 3 \wedge R$
2	3	4	$x \neq 2 \wedge y = 3 \wedge R$
x	y	1	$x = 2 \wedge y \neq 3 \wedge R$
2	3	2 4	$x = 2 \wedge y \neq 3 \wedge R$
x	y	1 2 4	$x = 2 \wedge y = 3 \wedge R$

$R = ((x < 0 \wedge t = 1) \vee (0 \leq x < 2 \wedge x + y = 2 \wedge t = 1))$

Table XIV
THE CONTRIBUTION OF THE FIRST R-BAG OF THE FIRST E-BAG TO THE RESULT OF $group_{A,B}(R)$

Algorithm 10 $group_{\bar{A}}(T)$

1: $V \leftarrow T$

2: Cluster the c-tuples of V into biggest bags of semi-unifiable c-tuples relative to $\bar{A}$ - we will call this *e–bags*. If a c-tuple belongs to more than one e-bag, then make copies of the c-tuple and put a copy in each e-bag. To do so, add the local condition $x = i$ to the i^{th} copy of the c-tuple for $i < u$ and the local condition $\bigwedge_{i=1}^{u-1} x \neq i$ to the u^{th} copy, where x is a newly introduced variable and u is the number of times the c-tuple is copied.

3: Partition each e-bag further into r-bags. To do so, call *break_up*$(\bigvee_{i=1}^{p} lc(t_i))$, where $\{t_i\}_{i=1}^{p}$ are the c-tuples in the e-bag that is being processed. This will produce a set of non-overlapping conjunctions $\{c_i\}_{i=1}^{w}$. Let $C = \{c_i\}_{i=1}^{w}$. Rewrite the local condition of each t_i as a disjunction of c_is. Next, break C into equivalence classes relative to the operation $\sim$. We define $c_i \sim c_j$ exactly when the set of the rewritten local conditions in which the two conjunctions appear is the same. Next, create an array D, where $D[i]$ is the disjunction of all the conjunctions in the i^{th} equivalence class. Reconstruct V by substituting each e-bag with a bag of *r-bags*. The c-tuples in i^{th} r-bag of a given e-bag will have the same local condition as the corresponding value of $D[i]$ and the main parts will correspond to the c-tuples that contained the local conditions that formed the equivalence class corresponding to $D[i]$.

4: From each r-bag, create a set of vertices, where each vertex corresponds to a distinct c-tuple in the r-bag (i.e., for duplicate c-tuples we will have a single vertex). Next, find all spanning undirected graphs that are transitive and have the property that if there is an edge between the vertices n_1 and n_3 and there exists a third vertex n_2 such that $t_1 \prec_{\bar{A}} t_2$ and $t_2 \prec_{\bar{A}} t_3$, where t_1, t_2 and t_3 are the c-tuples corresponding to the vertices, then there are edges between n_1 and n_2 and between n_2 and n_3.
Next, the set of nested c-tuples that correspond to each graph are created. Their union yields the result of doing the grouping. More precisely, suppose that we are examining an r-bag r and a graph G associated with it. Since G is transitive, it will contain a set of disjoint complete sub-graphs, where each such sub-graph will correspond to a resulting nested c-tuples. If the vertices in the complete sub-graph belong to the c-tuples $\{t_i\}_{i=1}^{p}$, then the corresponding nested c-tuple will have the single value $(x_1, \ldots, x_a)$ for the attributes $\bar{A}$, the bag of values $\{| \pi_{\bar{B}} main(t_i) |\}_{i=1}^{p}$ for the attributes $\bar{B}$, and the local condition $L_r \wedge R_G \wedge (\bigwedge_{i=1}^{p} [(\pi_{\bar{A}} main(t_i)) = (x_1, \ldots, x_a)])$. The condition L_r is the local condition of the r-bag r. The condition R_G is the condition that projection on the $\bar{A}$ attributes of the main parts of the c-tuples that correspond to nodes in G that are connected should be equal, while the projection on the $\bar{A}$ attributes of the main parts of the c-tuples that correspond to nodes in G that are not connected should be distinct.

polyhedra. Note that this operation is equivalence preserving. The additional constraint that all the conjunctions that form the $D[i]$ of a given r-bag appear in the same set of c-tuples' local conditions guarantees that the r-bags partition the possible valuations, that is, under every valuation the local condition of at most one r-bag of every e-bag will be true. In other words, given an arbitrary valuation and an e-bag of r-bags, either none of the c-tuples' local conditions will be true or the local conditions of all the c-tuples in exactly one r-bag will be true. For our example, Tables XII and XIII show the value of the C and D array, respectively. Note that, in order to keep the example simple, the local conditions are not normalized using the algorithm from [18].

Next, the algorithm constructs a set of graphs for each r-bag, where each graph corresponds to a valuation. In a graph, there is an edge between two vertices if under the corresponding valuation it is true that $\pi_{\bar{A}}(main(t')) = \pi_{\bar{A}}(main(t''))$, where t' and t'' are the c-tuples corresponding to the vertices. A graph is valid, that is, a corresponding valuation exists exactly when (1) the graph is transitive (2) if there is an edge between the vertices n_1 and n_3 and there exists a third vertex n_2 such that $n_1 \prec_{\bar{A}} n_2$ and $n_2 \prec_{\bar{A}} n_3$, then there are edges between n_1 and n_2 and between n_2 and n_3. This is why all the graphs having these two properties are constructed and these graphs show which c-tuples in the r-bag will be grouped relative to the attributes $\bar{A}$ under different valuations. Figure 2 shows the graph for the first r-bag of the first e-bag, where the c-tuple numbers are preserved from Table XI. The resulting c-tuples that are constructed from the four possible graphs are shown in Table XIV. ■

The following theorem proves the time complexity of Algorithm 10.

Theorem 21: A *variable c-tuple* is a c-tuple that has variables in it it, while a *regular c-tuple* is a c-tuple that does not. Let v be the number of variable c-tuples in T, m be the greater of the size of the longest c-tuple and the size of the global condition of T, n be the number of regular c-tuples with distinct main parts, r be the highest count of regular c-tuples that have the same main part but distinct local conditions, s be the number of attributes in T, and c be a constant. Then the total time to perform Algorithm 10

is $O((2^v+n)\cdot s+(2^v+n)\cdot 3^{\sqrt{2}^{m\cdot(v+r)}}\cdot(m\cdot(v+r))^c+(2^v+n)\cdot 2^{v+r}\cdot 2^{v+n})$.

Proof: Line 2 of Algorithm 10 takes $O(2^v+n)\cdot s$ time because it may take as much as $O(2^v\cdot s)$ time to partition the variable c-tuples and then $O(n\cdot s)$ time to determine the groups for the regular c-tuples. Note that we get this low time bound thanks to the fact that regular c-tuples with distinct main parts can not appear in the same e-bag.

Line 3 will take $O((2^v+n)\cdot 3^{\sqrt{2}^{m\cdot(v+r)}}\cdot(m\cdot(v+r))^c)$ time because the size of a c-tuple's local condition may grow to a size of $O(m\cdot(v+r))$ after the normalization procedure from [18] is applied.

Line 4 takes $O((2^v+n)\cdot 2^{v+r}\cdot 2^{v+n})$ time because there maybe as much as 2^{v+r} r-bags in each e-bag and each r-bag may contain as much as $n+v$ distinct c-tuples and therefore there are 2^{v+n} possible graphs for each r-bag. ■

B. Performing the Aggregation

Now that we have defined how grouping over c-tables can be done, performing aggregation is straightforward. The following definition contains the details.

Definition 18 (syntax and semantics of aggregation): Let T be a c-table. We will denote a grouping by the attributes $\bar{A}=\{A_i\}_{i=1}^a$ and aggregation for the attributes $\bar{B}=\{B_i\}_{i=1}^b$ as ${}_{A_1,\ldots,A_a}\mathcal{F}_{agg_1(B_1),\ldots,agg_b(B_b)}T$, where the sets $\bar{A}=A_1,\ldots,A_a$ and $\bar{B}=B_1,...,B_b$ are disjoint and their union yields all the attributes in T. The value for *agg* can be *min*, *max*, *sum*, *count*, or *avg*. Algorithm 11 shows the pseudo-code for performing the aggregation.

Algorithm 11 ${}_{A_1,\ldots,A_a}\mathcal{F}_{agg_1(B_1),\ldots,agg_b(B_b)}T$

1: $V\leftarrow group_{\bar{A}}T$
2: **for** t in V **do**
3: perform the mapping from Table XV to Table XVI on t, where $\{x_i\}_i=1^n$ are new variables and the function *con* is defined in Table XVII.
4: **end for**
5: **return** V

Note that Line 3 of Algorithm 11 performs aggregating over the $\bar{B}$ attributes by introducing new variables in the main parts of the result and moving the aggregations to the local conditions.

Theorem 22: The aggregation operator that is defined in Algorithm 11 is well defined.

Proof: The correctness of the grouping algorithm follows from Theorem 20 and the correctness of the *con* operator. Let us next examine the *con* operator. For the *min* operation it adds the condition that the new variable must be smaller than the value for the other c-tuples for that attribute in the group, which is the desirable behavior. The correctness of the *max* operation is analogous. The *count* operation is implemented correctly because it returns the count of the c-tuples in each group. The *sum* operation adds the condition that the value for the aggregate attribute must be equal to the sum of the values for that attribute in each group, which is the expected behavior. Finally, for the *avg* operator we add the condition that $x*n$ must be equal to the sum of the values for the aggregate attribute and therefore the new value for the attribute will be $\frac{\sum_{i=1}^{n} b_i}{n}$, which is exactly the average operator. ■

Table XVIII shows the result of ${}_B\mathcal{F}_{sum(A)}R_1$, where Table R_1 is defined in Table IV

A	B	*condition*
$a_1\ldots a_k$	$b_1^1\ldots b_n^1$	c
	$\ldots$	
	$b_1^p\ldots b_n^p$	

Table XV
A COMPLEX C-TUPLE t

A	B	*condition*
$a_1\ldots a_k$	$x_1\ldots x_n$	$c\wedge con(x_1,agg_1,b_1^1,\ldots,b_1^p)\wedge$ $\cdots\wedge con(x_n,agg_n,b_n^1,\ldots,b_n^p)$

Table XVI
THE RESULT OF ${}_{\bar{A}}\mathcal{F}_{agg_1(B_1),\ldots,agg_n(B_n)}(t)$

VI. CONCLUSION AND FUTURE RESEARCH

In the paper, we presented algorithms for querying c-tables extended with linear conditions using the closed world assumption. We have chosen this representation because it is the least expressive extension of c-tables over which bag relational algebra with grouping and aggregation is closed

(agg)	$con(x,agg,b_1,\ldots,b_n)$
min	$\bigwedge_{i=1}^{n}(x\le b_i)$
max	$\bigwedge_{i=1}^{n}(x\ge b_i)$
$count$	n
sum	$x=\sum_{i=1}^{n} b_i$
avg	$\underbrace{x+\cdots+x}_{n\text{ times}}=\sum_{i=1}^{n} b_i$

Table XVII
EXPLAINS THE OPERATOR *con*

A	B	*condition*
2	x	$x\neq 3\wedge(x\neq 4\vee$ FALSE$)$
2	4	TRUE $\wedge\ (x\neq 4\vee x=3)$
y	4	$x\neq 3\wedge$ TRUE $\wedge\ x=4\wedge y=2+2$

g.c. $x\neq 2$

Table XVIII
THE RESULT OF ${}_B\mathcal{F}_{sum(A)}R_1$

and can be well defined. As expected, the running time of the presented algorithms is polynomial relative to the size of the certain information and non-polynomial relative to the size of the incomplete information.

A major topic for future research is optimizing the algorithms for performing the different relational operations. For example, in the relational case, the join between two tables can be performed in different ways and the efficiency of performing the join depends on the implementation. The same applies for joining c-tables.

In general, there are different ways of performing the deferent relational algebra operations over c-tables. The purpose of this paper is to define their semantics by presenting example algorithms for doing the operations. The presented algorithms are not optimal and optimization techniques such as dynamic programming and iterative dynamic programming can be used to optimize the different relational algebra operators over c-tables.

Other possible extensions of the presented work follow.

• Explore c-tables with variables over additional domains, such as date or currency. • Speed up the presented algorithms by sacrificing their accuracy, that is, explore approximate query answering for incomplete information.
• Explore integrity constraints for incomplete information and how they can be used to perform semantic query optimization.
• Extend research done in relational databases, such as research on view maintenance, transaction control, logging and recovery, to databases that contain incomplete information.
• Explore introducing an ordering of the c-tuples in a c-table and defining an *order by* operator.

REFERENCES

[1] L. Stanchev, "Querying Incomplete Information using Bag Relational Algebra," *eKNOW 2010, Second International Conference on Information Process, and Knowledge Management*, pp. 110–119, 2010.

[2] T. Imielinski and W. Lipski, "Incomplete Information in Relational Databases," *Journal of Association of Computing*, vol. 31, no. 4, pp. 761–791, October 1984.

[3] L. Libkin and L. Wong, "Some Properties of Query Languages for Bags," *Proceedings of Database Programming Languages*, pp. 97–114, 1994.

[4] G. Grahne, *The problem of Incomplete Information in Relational Databases*. Berlin: Springer-Verlag, 1991.

[5] R. Reiter, "A Sound and Sometimes Complete Query Evaluation Algorithm for Relational Databases with Null Values," *JACM*, vol. 33, no. 2, pp. 349–370, 1986.

[6] L. Y. Yuan and D.-A. Chiang, "A sound and Complete Query Evaluation Algorithm for Relational Databases with Null Values," *ACM*, 1988.

[7] L. Libkin, "Query Language Primitives for Programming with Incomplete Databases," *Proceedings of DBPL*, 1995.

[8] P. Buneman, A. Jung, and A. Ohori, "Using Powerdomains to Generalize Relational Databases," *Theoretical Computer Science*, vol. 91, no. 1, 1991.

[9] R. Reiter, *On closed world databases, Logic and databases*. Plenum Press, 1978.

[10] G. Grahne, "Dependency Satisfaction in databases with Incomplete Information," *Proceedings of International Conference on Very Large Data Bases*, pp. 37–45, 1984.

[11] J. A. Biskup, "A Formal Approach to null Values in Database Relations," *Advances in Database Theory*, pp. 299–341, 1981.

[12] E. F. Codd, "Understanding Relations (Installment 7)," *FDT Bull. of ACM-SIGMOD*, vol. 3, no. 4, pp. 23–28, December 1975.

[13] ——, "Extending the Database Relational Model to Capture more Meaning," *ACM Transactions on Database Systems*, vol. 4, no. 4, pp. 397–434, December 1979.

[14] J. Grant, "Null values in Relational Data Base," *Information Processing Letters*, vol. 6, no. 5, pp. 156–157, October 1977.

[15] T. Imielinski and W. Lipski, "On Representing Incomplete Information in a Relational Data Base," *Proceedings of the 7th International Conference on Very Large Data Bases*, pp. 388–397, September 1981.

[16] M. Fischer and M. O. Rabin, "Super Exponential Complexity of Presburger Arithmetic," *Project MAC Tech. Mem. 43. MIT*, 1974.

[17] S.Basu, "New Results on Qunatifier Elimination over Real Closed Fields and Applications to Constraint Databases," *JACM*, vol. 46, no. 4, pp. 537–555, 1999.

[18] J. L. Lassez and K. McAloon, "Applications of a Canonical Form of Generalized Linear Constraints," *Journal of Symbolic Computation*, vol. 13, pp. 1–24, 1992.

[19] J. L. Lassez, T. Huynh, and K. McAloon, "Simplification and Elimination of Redundant Arithmetic Constraints," *Proceedings of NACLP*, 1989.

[20] A. Tarski, "A Decision Method for Elementary Algebra and Geometry," *University of California Press*, 1951.

[21] F. Jerrante and C. Rackoff, " A Decision Procedure for the First Order Theory of Real Addition with Order," *SIAM Journal of Computing*, vol. 4, no. 1, pp. 69–76, 1975.

[22] D. Kossmann and K. Stocker, "Iterative Dynamic Programming: A New Class of Query Optimization Algorithms," *ACM Transactions on Database Systems*, vol. 25, no. 1, 2000.

[23] G. Klir, U. Clar, and B. Yuan, *Fuzzy Set Theory. Foundations and Applications*. Prentice Hall, 1997.

[24] M. D. Springer, "The Algebra of Random Variables," *Wiley series in probability and mathematical statistics*, 1979.

[25] G. Kuper, L. Libkin, and J. Paredaens, *Constraint Databases*. Springer, 1998.

[26] G. M. Kuper, "Aggregation in Constraint Databases," *Proceedings of the 1st International Workshop on Principles and Practice of Constraint Programming*, pp. 161–172, 1993.

[27] J. Kleinberg, C. Papadimitriou, and P. Raghavan, "Auditing Boolean Attributes," *PODS*, pp. 86–91, 2000.

[28] A. Makinouchi, "A Consideration of Normal Form of Non-necessarily-normalized Relations in the Relational Data Model," *Proceedings. of International Conference on Very Large Data Bases*, pp. 447–453, 1977.