1 Aggregation in Constraint Databases

Gabriel M. Kuper

1.1 Abstract

We show how to add aggregation to a constraint database query language. One example of the use of aggregation in such a database is computing the area of a region in a geographic database. We show how aggregation can be added to the algebra and tuple calculus, and discuss the problems that arise from the interaction of aggregate operators and constraints.

1.2 Introduction

The idea of of combining relational databases with constraint formalisms is introduced in [KKR90] (see also [KG94] [Revesz90]). The key idea is the generalization of the notion of a tuple to a conjunction of constraints.

One important aspect of relational databases that is missing in these papers is a discussion of how aggregate operators interact with constraints. In a standard relational database, typical aggregate operators are: **sum**, **average**, **max** and **count**. In a constraint database, besides these operations, we also want to compute functions such **length**, **area** and **volume**. Other possible aggregate operators include **nearest** (finding the object in the database closest to a given point or object), and averaging an attribute (such as rainfall) over a given region.

In this paper, we outline a method to add aggregation to the languages of [KKR90]. Our approach generalizes the paper of Klug [Klug82] to constraint databases. The current paper has three main parts: We first show how Klug's approach can be generalized to the constraint framework, considering only the semantics of the database, not its representation. We show how to generalize the relational algebra and calculus under this assumption. We then examine the consequences of requiring a finite representation for the result (the *closure* property). Our results here are negative: We show that for the **area** aggregate operator, the closure property does not hold for any of the interesting classes of constraints, i.e., dense order, linear, or polynomial constraints. The third part of the paper is devoted to a discussion of ways around this problem. They include restricting the query language, restricting the class

of aggregate operators, and using a typed language to restrict the way aggregate operators can be used.

1.3 Constraint Databases

The classes of constraints that we consider in this paper are:

Definition 1

1. *Real polynomial inequality constraints* are formulas of the form

$$p(x_1, \ldots, x_j)\ \theta\ 0$$

where p is a polynomial with real coefficients and variables $x_1, \ldots, x_j$, and θ is one of $=, <, >$. The variables range over the domain D of real numbers.
2. *Dense linear order inequality constraints* are all formulas of the form $x\theta y$ and $x\theta c$, where x, y are variables, c is a constant, and θ is one of $=, <, >$. The variables range over a countably infinite set D (e.g., the rational numbers) with a binary relation $<$ that is a dense linear order.
3. *Linear inequality constraints* are all formulas of the form

$$f(x_1, \ldots, x_j)\ \theta\ 0$$

where f is a linear combination, with real coefficients, of the variables $x_1, \ldots, x_j$, and θ is one of $=, <, >$. The variables range over the set D of real numbers.

The basic definitions of constraint databases are (see [KKR90] for more details):

Definition 2 Let Φ be a class of constraints.

1. A *generalized k-tuple* (over variables $x_1, \ldots, x_k$) is a finite conjunction $\varphi_1 \wedge \cdots \wedge \varphi_N$, where each $\varphi_i, 1 \leq i \leq N$, is a constraint in Φ. Furthermore, the variables in each φ_i are all free and among $x_1, \ldots, x_k$.
2. A *generalized relation of arity k* is a finite set $r = \{\psi_1, \ldots, \psi_M\}$, where each $\psi_i, 1 \leq i \leq M$ is a generalized k-tuple over the same variables $x_1, \ldots, x_k$.

3. The *formula corresponding to a generalized relation* r is the disjunction $\psi_1 \vee \cdots \vee \psi_M$. We use ϕ_r to denote the quantifier-free formula corresponding to relation r.
4. A *generalized database* is a finite set of generalized relations.□

In database theory, a k-ary relation r is a finite set of k-tuples (or points in a k-dimensional space) and a database is a finite set of relations. However, the relational calculus and algebra can be developed without the finiteness assumption for relations. We will use the term *unrestricted relation* for finite or infinite sets of points in a k-dimensional space. In order to be able to do something useful with such unrestricted relations, we need a finite representation that we can manipulate. This is exactly what the generalized tuples provide.

Definition 3 Let Φ be a class of constraints interpreted over domain D, r a generalized relation of arity k with constraints in Φ, and let $\phi_r = \phi_r(x_1, \ldots, x_k)$ be the formula corresponding to r with free variables $x_1, \ldots, x_k$. The generalized relation r represents the unrestricted k-ary relation which consists of all $(a_1, \ldots, a_k)$ in D^k such that $\phi_r(a_1, \ldots, a_k)$ is true. A generalized database represents the finite set of unrestricted relations that are represented by its generalized relations. □

In [KKR90] the relational tuple calculus is extended to generalized relations. This language is the same as the language we describe below, if we have no aggregate operators.

In the next two sections, we shall discuss the extension of the relational algebra and calculus with aggregation to unrestricted relations, i.e., we ignore, for now, the issue of representation of the result.

1.4 The Algebra

In [Klug82], for each aggregate function, e.g., **sum**, we have a parameterized family of **sum** functions

$$\mathbf{sum}_1, \mathbf{sum}_2, \ldots, \mathbf{sum}_i, \ldots$$

$\mathbf{sum}_i$ sums the numbers in the ith column of its input, taking duplicates into account.

Aggregate functions such as **sum**, **count**, **average** are still available in our language, but they are defined only when the corresponding

instances are finite. We also have aggregate functions such as **area**, **volume**, etc. Note that **area** is a *binary* function, and we therefore need to have a family of functions $\{\mathbf{area}_{i,j}\}$, where $\mathbf{area}_{i,j}$ computes the area of the set of points described by the ith and jth columns of the relation.

Note that this set of points can contain duplicates. The idea behind Klug's approach is to let the user specify whether duplicates should or should not be considered when applying the aggregate functions. We can allow this in our language by assigning a number to each point in the region, representing how many times it occurs. Under appropriate measurability assumptions (that hold for all the classes of constraints defined above), we can define the area as the integral of this function [Halmos50].

Some care is needed to ensure that queries are safe. Finite aggregation functions (such as **sum**) can only be applied when the corresponding relation is finite, and infinite functions (such as area) can only be applied when the result is well-defined (in this case, when the region is bounded). This is a topic that needs more research: For now, we shall assume that all queries, and all their subqueries, are well-defined. Our results show that the calculus/algebra equivalence holds for all queries that satisfy this assumption.

We now describe the algebra formally. We have a set

$$\mathrm{Agg} = \{f_\alpha, f_\beta, \ldots\}$$

of aggregate functions. Each function is from the set of all relations to the domain D of the constraints. We need a "uniformness" condition, that basically says that whenever we have an aggregate function, such as $\mathbf{area}_{1,2}$ in Agg, we also have "similar" functions $\mathbf{area}_{i,j}$, for all $i \neq j$, that operate on all other possible sets of columns.

Definition 4 Agg has the following property. For every $f \in Agg$, and every relation S, if S' is a constant expansion of S, i.e., if there is a projection such that $\Pi_X(S') = S$ and $\Pi_{\neg X}(S')$ contains exactly one tuple, then there is a $f' \in Agg$ such that $f(S) = f'(S')$.

Definition 5 The relational algebra is defined in exactly the same way as for languages without aggregation, with the following additional rule.

- If e is a relational expression of arity n, X a sublist of e's attributes, and $f \in \mathrm{Agg}$, then $\mathrm{e}\,\langle X, f\rangle$ is a relational expression of arity $n+1$.

The semantics of such an expression are defined as follows.

Definition 6 Let e be a relational expression of arity n, let X, f be as above and let I be an instance. Then

$$e\,\langle X, f\rangle\,(I) = \{t[X] \circ y \mid t \in e(I) \wedge y = f(\{t' \mid t' \in e(I) \wedge t'[X] = t[X]\})\}$$

Note that f is applied to a set of tuples, rather than a multiset, and therefore that all duplicate *tuples* are removed. All treatment of duplicates is "hidden" in the definitions of the aggregate functions, since the set of columns on which the aggregation is actually performed may contain duplicates.

We illustrate the algebra by several examples, based on the rectangle example in [KKR93].

Example 1 Let $R(z, x, y)$ be a ternary relation that represents a set of rectangles. We interpret a tuple (z, x, y) in R to mean that (x, y) is a point in the rectangle with name z.

There are various ways in which we could apply the **area** function to this relation.

- Suppose we want to compute the area of each rectangle separately. In other words, we want to compute a relation $S(n, a)$, where a is the area of the rectangle with name n. This can be computed by the query

$$R\,\langle 1, \mathbf{area}_{2,3}\rangle$$

i.e., for each z we compute the area of the set of points with name z.

- Now suppose that we are interested in the total area covered by the points of the database, counting points that are in several rectangles only once. We can do this by projecting R onto the second and third columns, thus generating a flat representation of the data, without the rectangle names. We then apply the aggregate operator to the result. Formally, the query is:

$$(\Pi_{2,3}(R))\,\langle \emptyset, \mathbf{area}_{1,2}\rangle$$

- Finally, suppose that we want to compute the total area, but this time counting points that are in several rectangles multiple times. This is simply

$$R\,\langle \emptyset, \mathbf{area}_{2,3} \rangle$$

since the aggregate operator is now applied without eliminating duplicate points. An alternative way to express this query would be to apply the **sum** operator to the result of the first example, i.e.,

$$R\,\langle 1, \mathbf{area}_{2,3} \rangle\,\langle \emptyset, \mathbf{sum}_2 \rangle$$

We should mention that even though this example is of a database of rectangles, the queries will also work for a database that consists of any type of geometric objects for which the **area** function is well-defined.

1.5 The Calculus

The definition of the calculus with aggregation is significantly more complicated than that of the relational calculus. Basically, the reason for this is that the relational calculus uses quantification only on the base relations. We, on the other hand, have to be able to quantify over relations that are the result of an aggregate operation, and the only way to do this is to actually compute the result. The calculus, therefore, has to have a way to construct intermediate relations.

For this reason, Klug's tuple calculus is defined via mutual recursion on 3 type of expressions: terms, formulas, and alphas. Closed alphas are the "top level", i.e., the queries themselves.

Definition 7 A term is one of

- A constant
- $x.A$, where x is a (tuple) variable and A a column number
- $f(\alpha)$, where α is an alpha

Definition 8 A formula is one of

- $\alpha(x)$, where α is a closed alpha, and x is a variable
- $\mu(t_, \ldots, t_k)$ where μ is a constraint and $t_1, \ldots, t_k$ are terms
- Disjunctions and negation of formulas
- $(\exists r)\psi$, where ψ is a formula, and r a range formula.

Range formulas are defined as follows.

Definition 9 Let x be a variable and let $\alpha_1, \ldots, \alpha_k$ be closed alphas. Then

$$\alpha_1(x) \vee \cdots \vee \alpha_k(x)$$

is a range formula.

Such a formula means that x is is in the union of the relations defined by the α_i's. Finally, we define alphas.

Definition 10

- For each relation R symbol of arity i, R is an alpha of degree i.
- If $t_1, \ldots, t_n$ are terms, $r_1, \ldots, r_m$ are range formulas for the free variables in these terms, and ψ is a formula, then

$$(t_1, \ldots, t_n) : r_1, \ldots, r_m : \psi$$

is an alpha of degree n.

Free variables, closed objects, etc. are then defined as usual. The one subtle point is that a variable can be bound by a quantifier, or by being in the range fragment $(r_1, \ldots, r_m)$ of an alpha. In the latter case, variables in $t_1, \ldots, t_n$ also become bound.

We will not describe the semantics here; the corresponding definitions of [Klug82] carry over with no problems.

Example 2 We show how to express calculus queries equivalent to the algebraic queries in the previous example.

- Compute the area of each rectangle separately.

$$(x.1, (\mathbf{area}_{1,1}(y.2, y.3) : R(y) : y.1 = x.1)) : R(x) : -$$

- Total area, without duplications.

$$(\mathbf{area}_{1,2}((x.2, x.3) : R(x) : -)) : - : -$$

- Total area, with duplications.

$$(\mathbf{area}_{2,3}(R)) : - : -$$

The following example illustrates the advantages of the calculus over the algebra.

Example 3 Consider a relation R that stores information about rainfall. R is a ternary relation, where (r, x, y) in R means that the total rainfall at point (x, y) was r. Suppose that we want to compute, for each r, the area of the region with total rainfall $\geq r$. This could be done by the following query

$$(x.1, \mathbf{area}_{1,2}((y.2, y.3) : R(y) : y.1 \geq x.1)) : R(x) : -$$

It would be much harder to express this query directly in the algebra.

In the result of this query, the first column only contains values that actually appear in the database. To be able to look up arbitrary values, we use the following query

$$(x, \mathbf{area}_{1,2}((y.2, y.3) : R(y) : y.1 \geq x)) : D(x) : -$$

where $D(x)$ means that x is in the domain D.

We can now show that the algebra and calculus are equivalent. Klug's proof works in our setting—the only things that have to be verified are that the proof does not depend on finiteness of the relations, or on the fact that aggregate functions have only one subscript. Note that we make rather stringent safety requirements, i.e., we show equivalence only between queries that are safe, and all of whose subqueries are safe. Further research is needed to see whether these requirements can be relaxed.

1.6 Closure under Aggregate Operations

So far, we have discussed the relational algebra and calculus on unrestricted relations. In constraint databases, we require that each relation instance be representable as a generalized relation. This means that the result of applying an aggregate operator to a generalized relation, must be a generalized relation. In other words, the result must be representable by the same class of constraints.

In this section, we show that for all the classes of constraints under consideration, this is false.

1.6.1 Linear Constraints

Let $R(x, y, z)$ be a ternary relation over the class of linear constraints,and suppose we want to compute, for each z, the area of the corresponding set of (x, y) points in R. This means that we want to evaluate the calculus query

$$(x.3, \mathbf{area}_{1,2}((y.1, y.2) : R(y) : y.3 = x.3)) : R(x) : -$$

or the (equivalent) algebraic query

$$R\,\langle 3, \mathbf{area}_{1,2}\rangle$$

Consider the following instance I of R

$$I = \{x - y - z \leq 0 \wedge x \geq 0 \wedge y \geq 0\}$$

The result of the query is a binary relation $S(z, t)$, where $S(z, t)$ holds iff $z^2 = 2t$, and this cannot be expressed using linear constraints.

Intuitively, this should not be too surprising. Computing are is basically the same as integrating a function—and of course the integral of a linear function is a quadratic one. It turns out, less intuitively, that the same problem arises for dense linear order and polynomial constraints.

1.6.2 Dense Linear Order

As before, let $R(x, y, z)$ be a ternary relation, with the same query,

$$R\,\langle 3, \mathbf{area}_{1,2}\rangle$$

Let I be the instance

$$I = \{0 < x < y < z < 1\}$$

The result once again is the binary relation $S(z, t)$ where $S(z, t)$ holds iff $z^2 = 2t$, and this cannot be expressed as a generalized relation using order constraints.

1.6.3 Polynomial Constraints

Let R be as before, with the same query $R\,\langle 3, \mathbf{area}_{1,2}\rangle$. Let I now be the instance

$$I = \left\{ \begin{array}{l} x^2 + y^2 \leq 1 \\ x \leq z \\ x, y, z \geq 0 \end{array} \right\}$$

The result is a relation $S(z,t)$ where $R(z,t)$ holds iff t is the area of the part of the unit circle that is above the x-axis, and between the lines $y = 0$ and $y = 1 - z$. This means that $S(z,t)$ holds iff

$$t = \int_0^{1-z} \sqrt{1-x^2}dx$$

or

$$t = \frac{1}{2}\left((1-z)\sqrt{z(2-z)} + \sin^{-1}(1-z)\right)$$

This cannot be expressed by polynomial constraints.

1.7 Attempts to deal with the closure problem

So far, we have seen that we can add aggregation to the query language with unrestricted relations, but that if we try to add aggregation to generalized relations we immediately encounter closure problems. This is clearly unsatisfactory. We describe three approaches to dealing with this problem: restricting the query language, restricting the aggregate operators, and restricting the interaction between the aggregation and the rest of the database, via typing.

1.7.1 Restricting the query language

It may well be the case, that for many applications, the full power of the relational query language with aggregation is not needed. For example, it might suffice to be able to apply the **area** operator as the final step of a query, without being interested about how the area varies with respect to some other parameter. In such a case the result would be a single number, or set of numbers, and no problems would arise.

We shall not discuss this approach in any more detail here. This approach will probably involve ad. hoc. decisions for specific applications, and at the present time it is not clear what general rules dhould be used to design such a query language. Some of the discussion in Scetion 6.3 is relevant to this approach as well.

1.7.2 Restricting the aggregation operators

The negative results in the previous section were for a specific aggregate operator **area**. A natural question to ask is whether there are any ag-

gregate operators for which the corresponding query language is closed. In turns out that for the **max** operator, the language is indeed closed.

We outline a proof that the algebra with the **max** operator, over the domain of dense linear order constraints is closed. In fact, we actually show more than this. We show that the **max** operator can be defined directly in the language. This of course, immediately implies that the language with **max** is closed.

For example, suppose that R is a ternary relation. The query

$$R\langle 3, \mathbf{max}_2\rangle$$

is equivalent to the (calculus query,

$$\{(z,n) \mid (\exists x)R(x,n,z) \wedge (\forall x,y)(R(x,y,z) \rightarrow y \leq n))\}$$

and this is a relational calculus query without aggregation. It is not hard to generalize this to arbitrary relational queries with **max**.

An interesting problem is to characterize, for different classes of constraints, which aggregate operators satisfy the closure property. In particualr, it is an open problem whether the query language is closed under *any* interesting aggregate operator that strictly increases the expressive power of the language. Some work in this direction is described in [CK94].

1.7.3 Using a typed language

Consider the **area** function. Up to now, we have assumed that **area** is a function from $\mathcal{P}(R \times R)$ to R. If we consider the physical units of the domain and range of **area**, we see that they are in fact different. Since arbitrary constraints between arguments that represent different physical dimensions have no physical meaning, there is no goo reason to allow such constraints in our language.

We propose that, by appropriate typing of the domains, we limit the way in which aggregate operators can be applied. This enables us to express many interesting queries, without the closure problem arising.

We illustrate this by an example. Consider a typed language with three types τ_1, τ_2, and τ_3. The interpretation of these types will be

- τ_1: Length. Constraints on variables of type τ_1 are polynomial constraints

- τ_2: Time. Constraints on variables of type τ_1 are dense linear order constraints
- τ_3: Area. Constraints on variables of type τ_1 are linear constraints

The **area** function is of type $2^{\tau_1 \times \tau_1} \rightarrow \tau_3$.

We do not allow arbitrary application of **area**–the set Agg contains functions $\mathbf{area}_{i,j}$ only when columns i and j are of type τ_1 and the remaining columns are of type τ_2.

One example where such a language could be used is in a land use database. In such a database, the boundaries of the spatial regions change only at discrete points of time, and area is computed only relative to this parameter. On the other hand, in a meterological database, where the data varies continuously, this model would be inappropriate. We can then define an algebraic query language using these restricted aggregate functions, and this language will then be closed.

This has just been a rough sketch. More work is needed to formalize such a notion of typed language, to classify which typed languages and aggregate functions yield closed languages, and to define a tuple calculus equivalent to this typed algebra.

Acknowledgments

I would like to thank Paris Kanellakis for useful discussions, in particular for providing some of the motivation for this paper.

Bibliography

[CK94] J. Chomicki and G. M. Kuper, Measuring Infinite Relations. *in preparation.*

[Dincbas88] M. Dincbas, et al. The Constraint Logic Programming Language CHIP. *Proc. Fifth Generation Computer Systems*, Tokyo Japan, 1988.

[Halmos50] P. Halmos. Measure Theory. *Van Nostrand Reinhold*, 1950.

[JL87] J. Jaffar, J.L. Lassez. Constraint Logic Programming. *Proc. 14th ACM POPL*, 111–119, 1987.

[KG94] P. C. Kanellakis and D. Q. Goldin. Constraint programming and database query languages. In *Proc. 2nd Conference on Theoretical Aspects of Computer Software (TACS)*, April 1994. (To appear in LNCS Spring-Verlag volume).

[KKR90] P. Kanellakis, G. Kuper and P. Revesz. Constraint Query Languages. *Proc. 9th ACM PODS*, pp. 299–313, 1990.

[KKR93] P. Kanellakis, G. Kuper and P. Revesz. Constraint Query Languages. JCSS, to appear.

[Klug82] A. Klug. Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions. *J. ACM* 29 (3), pp. 699–717, 1982.

[Revesz90] P.Z. Revesz. A Closed Form for Datalog Queries with Integer Order. *Proc. 3rd International Conference on Database Theory*, 1990.

[VanHen89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.