An Informal Introduction to Constraint Database Systems

Volker Gaede
CSIRO Division of IT
GPO Box 664
Canberra ACT 2601
Australia
gaede@cbr.dit.csiro.au

Mark Wallace
IC-Parc, William Penney Laboratory
Imperial College, London, SW7 2AC
UK
mgw@doc.ic.ac.uk

Abstract

In this paper, we provide an informal introduction to constraint database systems, a new database paradigm which has significant advantages over other database models such the object-oriented or relational one. In fact, constraint database systems can integrate the most prominent features of other models within a single framework based on constraints while overcoming some of their problems. At the beginning, we motivate the use of constraints in databases by arguing that they provide a high level language for describing problems. Next, we give a short introduction to constraint programming, which gave rise to research into constraint database systems. Before turning our attention to recent research contributions and open issues, we review some of the most important constraint domains along with a wide range of applications areas where they are deployed.

Keywords: constraint database system, constraint programming, constraint logic programming, multidimensional access methods, advanced databases.

Contents

1 Introduction **2**
1.1 Generalized Tuples 2
1.2 Data Modeling 4
1.3 Optimization 5

2 Constraint Programming **8**
2.1 Logic and Databases 8
2.2 The Constraint Logic Programming Scheme 9
2.3 Concurrent Constraints 11

3 **Application Areas** **13**
3.1 From Constraint Programming to Constraint Databases . . . 13
3.1.1 Scaling Up Combinatorial Problems? 13
3.1.2 Loose and Tight Database Couplings 14
3.1.3 Persistent Constraint Stores 16
3.2 Constraint on the Real Numbers 16
3.2.1 Engineering 17
3.2.2 Medicine 18
3.2.3 GIS and Environmental Systems 19
3.2.4 Temporal Databases 20
3.2.5 Financial Systems 21
3.3 Finite Domain Constraints 21
3.4 Boolean Constraints 22
3.5 Set Constraints 23
3.6 Dynamically Changing Constraints 23

4 **Theoretical and Practical Issues** **24**
4.1 Database Queries and Complexity 25
4.1.1 Expressive Power and Data Complexity 25
4.1.2 Query Languages 25
4.1.3 Complexity Classes 26
4.2 Query Complexity 27
4.3 Expressive Power 28
4.3.1 Infinite Queries 28
4.3.2 Generic Queries 29
4.3.3 Aggregate Queries 30
4.4 Indexing 30
4.5 Query Optimization and Processing 32
4.6 Object-Oriented Features 34
4.7 Systems 34

5 **Summary and Future Directions** **36**

1 Introduction

Constraint database systems represent a new paradigm in the area of database systems in that they unify and extend several distinct fields of research. Specifically, these include relational, object-oriented, spatial and temporal databases; geographic information systems (GIS); and constraint logic programming (CLP).

1.1 Generalized Tuples

One key idea of constraint databases (CDB, for short) is to view data as constraints and constraints as data. Despite its simplicity, this idea opens up some big horizons and offers cleaner, more powerful data models, a generic representation of different kinds of data, and optimized query evaluation.

However, before we start presenting some of the most recent results, let us illustrate the idea with an example. Suppose that we want to store some personnel data in a database system. In a relational model, we do so by considering every person as a tuple (after having normalized our data schema to, say, third normal form). As an example consider the following tuple

$$(\text{`Bruce}', \text{`Willis}', \text{`GroupLeader}', 01/5/90)$$

which represents Bruce Willis, his current position and the date he attained this position. Similarly, we could represent Bruce in an object-oriented or deductive data model. However, let us now have a look at the following representation where $\wedge$ denotes the and-connective.

$$\{(\mathsf{FN}, \mathsf{LN}, \mathsf{Pos}, \mathsf{Date})| \quad \mathsf{FN} = \text{`Bruce}' \wedge \mathsf{LN} = \text{`Willis}' \wedge \mathsf{Pos} = \text{`GroupLeader}' \wedge \mathsf{Date} = \mathsf{01/5/90}\}$$

This formula states that the person's first name (FN) is Bruce, the last name (LN) is Willis and so on, by saying that the attribute first-name (represented by the variable[1] FN) has to be equal to some given value (constant). Each equation, e.g. $FN = \text{`}Bruce'$, is termed an atomic *constraint*, and a conjunction of constraints is also considered to be a constraint. Consequently the above formula is also a constraint.

Clearly, the above constraint also identifies the person we are interested in, but exemplifies a slightly different view. This view of data tuples was first put forward by Kanellakis, Kuper, and Revesz (1990) in a seminal paper which proposed a framework for constraint database systems by extending the idea of a tuple. Their *generalized tuple* has the form $p_j(X_1, ..., X_k)|C$ with C being a constraint on the attributes X_i and k being the *arity* of the generalized tuple. Obviously, this view encloses relational database systems as a special case. A traditional relation of with n attributes is a finite set of generalized tuples of arity n whose atomic constraints are all equations, as in the above example.

The first generalization made possible by constraints is to use a single constraint to represent a whole set of values. Using inequality constraints it is possible to adapt the above example to represent the whole time interval (set of dates) during which Bruce Willis was a group-leader in a concise manner:

$$\{(\mathrm{FN}, \mathrm{LN}, \mathrm{Pos}, \mathrm{Date})| \quad \mathsf{FN} = '\mathrm{Bruce}' \wedge \mathsf{LN} = '\mathrm{Willis}' \wedge \mathsf{Pos} = '\mathrm{GroupLeader}' \wedge \mathsf{01/01/95} \geq \mathsf{Date} \wedge \mathsf{Date} \geq \mathsf{01/5/90}\}$$

In the relational model the above generalized tuple would correspond to a large number of tuples.

[1] In this paper we shall henceforth use "variable" and "attribute" interchangeably.

The second generalization with respect to other database models is that a single constraint can involve more than one attribute (variable). To illustrate this, let us assume that we want to store for each item in a warehouse also its three dimensional shape. Instead of representing as a set of points with unclear semantics, the concept of generalized tuples allows us to record the actual three-dimensional volume in a much cleaner way. As an example, the following tuple represents the wedge-shaped item "P123":

$$\{(\mathrm{Item}, \mathrm{X}, \mathrm{Y}, \mathrm{Z}) | \quad \mathsf{Item} = {'}\mathrm{P123}{'} \wedge 10 \geq \mathsf{X} \wedge \mathsf{X} \geq 0 \wedge 10 \geq \mathsf{Y} + \mathsf{Z} \wedge \mathsf{Y} \geq 0 \wedge \mathsf{Z} \geq 0\}$$

In this case the underlying domains of X, Y and Z are continuous, so the generalized tuple represents an infinite set of data points.

Although we refer to the data item as being a "generalized tuples", it should be clear that deductive and object data models can be generalized in exactly the same way as the relational model (Brodsky and Kornatzky 1995). In a constraint object database, instead of an attribute having a single value (which corresponds to the equation $Attribute = Value$), the attribute can be specified in terms of constraints (e.g. $Attribute \geq Value$). More generally, the constraint may involve more than one attribute of the object, just as for generalized tuples. As should have become clear from the above examples, a single constraint database relation can naturally accommodate alphanumeric, temporal, and spatial data.

1.2 Data Modeling

Increasingly databases are required to store heterogeneous data. Currently geographic data, spatial data, and CAD/CAM data must often be held in separate databases or even in file systems, since the favored database is not capable of dealing efficiently with heterogeneous data. Therefore, it is typically the task of the respective application program to bridge the gap between the different data stores by means of interfaces. Obviously, such a separation makes applications harder to build, and it also allows inconsistencies to creep in, because each transaction involves multiple data sources. It also prevents various optimizations, sometimes imposing unacceptable penalties on certain kinds of query.

Fortunately, generalized tuples have the potential to capture all these different kinds of data within a single framework. Moreover, this representation is a very natural one, since constraints are the normal language of discourse at the application level. Typically, alphanumeric data is captured as equality constraints, spatial and temporal data as linear constraints. More complex constraints, involving polynomial equations and inequations, can also be expressed as generalized tuples, if necessary to support sophisticated CAD/CAM applications.

Generalized tuples provide a very abstract data representation. In current databases the same data may have different possible concrete representations. For example, in geographic information systems two-dimensional points can be represented using either a vector (X, Y) or an angle and a distance (θ,R) with respect to some unique origin. In case data comes from two different databases, it may happen that the two different databases use different representations and origins. These representations must be translated appropriately, but within the database there is nothing to prevent the pair (X, Y) from being interpreted as an angle and a distance. Since generalized tuples do not have a hidden semantics, they can be used as a common representation for transactions involving multiple, federated or distributed, databases. Constraints can be further used to specify the relationship between different sources. Thus, constraint database systems can be seen as a higher level data model supporting a unified view of the underlying data.

Brodsky (1996) points out that constraints as data contributes to *uniformity*, expressiveness, compactness and optimizability. So far we have emphasized uniformity, thus let us briefly outline by means of an example how their expressiveness and compactness contribute directly to high level, natural data modeling and how constraints enhance an object-oriented data model for storing spatial data.

Assume that we have to fit out a room with furniture such a desk. Let us further assume that the room as well as the furniture have some attributes representing their shape. Within the framework of a spatial object-oriented data model with constraints (Hermosilla and Kuper 1995), the shape of a desk could be expressed as follows:

```
desk1 is_obj [shape= {(x1,y1)| -4=<x1=<4 & -2=<y1=<2}, ... ]
```

With this data model it is easy to ask where in `room1` the desk can be placed so it fits completely inside the room:

```
[A] [room1[shape=R] & A subset R & desk1[shape=D] & (A+D) subset R]
```

`A` is the result variable, whose feasible values are possible locations for the middle of the desk. `D` ranges over all points of the desk, represented as displacements from its middle. Clearly the feasible values of `A` are a subset of the points in the room `R`. Moreover, each point of the desk must also be in the room `(A+D) subset R`

1.3 Optimization

The little advertised drawback of high-level data models is that apparently simple queries often require an amazingly long time to be answered. The mapping of such a data model to a particular physical data structure may support highly efficient evaluation for some queries, but it will typically penalize others that appear equally simple.

Constraint databases not only admit optimization, but actually support optimized query evaluation. Typically complex data models (e.g., GIS models), admit a multitude of operators, such as polygon intersection, inclusion, and disjointness. For certain applications extra operators are required: e.g. to generate a polygon inclosing all points within a certain distance of the input polygon; or to return a polygon to the north-west of the current polygon.

When a variety of such operators are used in a query, the optimizer can only treat them separately. Even if it has the necessary information to apply the operator at the optimal point in the query execution (Gaede and Günther 1995a), it cannot handle the interaction between the different operators.

In a constraint database, the operators can all be expressed as constraints over the given constraint domain (see the following section for the definition of a constraint domain). A constraint solver, which is a specialized piece of software tuned to efficiently handle sets (i.e. conjunctions) of constraints over the domain is associated with each domain. Consequently, the different operators involved in a query are handled together by the constraint solver so as to achieve efficient query evaluation even for highly complex queries (Brodsky, Segal, and Exarkhopoulou 1996). This is a clear advantage over object-oriented and relational database system, which resort to some imperative programming language alien to the database query optimizer for implementing such functionality. Thus, constraint database systems have the potential to overcome the impedance mismatch not just hiding it.

Note also that a constraint solver is often capable of identifying inconsistencies among the constraints without even instantiating them, that is, without binding the variables to values. As an example, consider the conjunction of constraints $X < Y, Y > X$. Obviously, there are no possible values for X and Y satisfying these constraints, and a corresponding solver will detect this immediately. However, at this point it is not yet clear how to integrate constraint solvers with the set-at-a-time model of database systems. One possible way is that the query execution algorithm posts all constraints to the solver immediately. Then as query processing proceeds more constraints on the query variables are accumulated, until either the query completes or an inconsistency is detected, that is, the associate tuple does not belong to the result set.

Moreover the constraints can be used by the optimizer not only passively - passing them to the constraint solver to check for satisfiability - but also actively (Bressan 1994; Miranker, Bayardo, and Samolados 1996). The active use of constraints is often called constraint *propagation*. An example of constraint propagation is in handling the "north west" operator. Clearly when searching for polygons northwest of a given point (x_0, y_0), it suffices to search in the range $(X \leq x_0,\ Y \geq y_0)$. Thus in the query "find points (X, Y) northwest of (x_0, y_0)", the constraints $(X \leq x_0)$ and $(Y \geq y_0)$ are propagated.

Constraint propagation extracts from a constraint, or a set of constraints, any information that may be useful in optimizing the handling of the remainder of the query. Sometimes this information may be weaker than the actual constraint. For example in handling a distance constraint, "find points (X, Y) with a distance d of the point (x_0, y_0)", the propagated information may be $(x_0 - d \leq X \leq x_0 + d)$ and $(y_0 - d \leq Y \leq y_0 + d)$.

Propagating on a set of constraints is, typically, much more powerful than propagating on a single constraint. For example to find points (X, Y) northwest and within a distance d of (x_0, y_0), the information propagated is

$(x_0 - d \leq X \leq x_0)$ and $(y_0 \leq Y \leq y_0 + d)$.

Of course the active use of constraints really pays off if the information propagated can be used in combination with an existing index so that only relevant data items are retrieved. Traditionally indices have been designed to support only two kinds of query conditions: equality and ranges (i.e. $<$ and $>$). However in database systems storing more complex data, there are other kinds of constraints which may appear as conditions in queries - or indeed produced as a result of constraint propagation. Thus GIS, temporal, spatial and CAD/CAM databases require new kinds of indexing.

Indices in constraint databases also store constraints. For a hierarchical index, each node records the (most specific) constraint satisfied by all the data items under that node. Constraints are accumulated throughout query evaluation, both during progression down the index tree and when data is retrieved. As constraints are accumulated they are tested for consistency and used to propagate further constraints, which can be used in optimizing the remainder of the query. Constraint handling can be highly tuned for the actual class of constraints occurring in the query, the index, and the data.

Since index entries, data items (i.e., *generalized tuples*), as well as the query conditions are expressed in the uniform framework of constraints, the query optimizer has obviously a much better handle on finding an efficient query evaluation strategy than in other database models. Consequently fully general and coherent database query optimization is both possible and natural.

Before completing the introduction, we should further mention two very important uses of constraints in databases. The first is *integrity constraints*, which are constraints which must be satisfied in every database state. For stored constraints $(X, Y)|C$, it is the case that *every* pair (X, Y) satisfying the constraints belongs to the database. This is not the case for integrity constraints. Although integrity constraints in databases have been studied for some years (Kowalski, Sadri, and Soper 1987; Wallace 1991), it is not clear how to integrate them seamlessly into database systems other than constraint-based ones.

While the technology of generalized tuples and integrity constraints are both interesting and important, they are in fact orthogonal, and we therefore have decided to minimize any confusion by leaving out integrity constraints.

The second is to use constraints for capturing *imprecise (partial) information* about objects. As before the semantics of the constraints changes, depending on whether we consider them as being generalized tuples or imprecise information. For example, if we think of the constraint $1 < X < 10$ as a generalized tuple stored in a relation r, then the constraint holds for *every* value of X in the range $1 - 10$. However, viewed as incomplete information, $1 < X < 10$ would only say that there exists *some* value of X in the range $1 - 10$ for which r holds.

There has been interesting work on the use of constraints in databases to record incomplete information, for example (Koubarakis 1994), however since the required technologies are (largely) orthogonal, we have decided to avoid a discussion of partial information and indefinite databases.

The remainder of this text is organized as follows. In Section 2, we give a short introduction to constraint programming, which gave rise to research into constraint database systems. Before turning our attention to recent research contributions in Section 3, we review some of the most important constraint domains along with a wide range of applications areas in Section 4.

2 Constraint Programming

2.1 Logic and Databases

The concept of a constraint database presented by Kanellakis, Kuper, and Revesz (1990) was inspired by the Constraint Logic Programming (CLP) Scheme of Jaffar and Lassez (1987). In this section we introduce the CLP Scheme, show how it relates to databases, and explore some important variants and extensions.

In the 1960's the relationship between logic and computing was an important research focus and many automatic theorem proving systems were built. It was recognized that general theorem provers could not be made efficient enough for most practical purposes, so some restricted classes of formulae were identified that could be handled in an efficient way. On this basis, several important computing languages and systems were developed. One of these was the relational database (Codd 1970) and another was logic programming (van Emden and Kowalski 1976). These two approaches have continued to influence each other over the years and so the migration of constraints from logic programming to databases is a natural phenomenon.

In a relational database each (traditional) tuple stores a fact, which in predicate logic is termed an *atom*, e.g.

emp('Bruce','Willis','Group Leader',01/5/90)

emp is termed the *predicate* and *'Bruce'*, etc. are its *arguments*. A query, when expressed in the relational calculus, is simply a predicate calculus formula. Its evaluation corresponds to a proof of this formula, using all the atoms in the database as axioms.

While theorem proving in general is only semi-decidable, evaluating relational calculus queries against a database has a complexity polynomial in the size of the database and the number of quantifiers in the query. Admitting recursion in queries adds dramatically to the worst-case complexity (Chandra and Harel 1980).

After some thirty years, the restriction that relational databases hold only atomic facts is being lifted. One direction is the move towards object databases, where the predicate calculus underpinnings of the relational model are replaced with a higher order logic (**?**; **?**). In practice this simply means that no sharp distinction is made between *predicates* and *functions*.

The other direction, which is taken in active databases, deductive databases and constraint databases, is to allow not only tuples to be stored but also queries or procedures. Translated into logic, this means that not only atomic facts are stored, but also complex formulae. In active and deductive databases these may be arbitrarily complex, but in constraint database the generalized tuples are restricted to a fixed set of predicate and function symbols and a very restricted structure.

We emphasize that the two directions are orthogonal and fully compatible: if we can specify an object database and we can specify a constraint database, then we can specify an object constraint database as well.

This second direction brings relational databases a step closer to logic programming, which allows a very restricted class of queries, but quite powerful stored information, in the form of program clauses.

2.2 The Constraint Logic Programming Scheme

Logic programming handles a restricted class of formulae called *Horn clauses*, which are disjunctions of literals in which at most one literal is positive: $P_1 \vee \neg P_2 \vee \ldots \vee \neg P_n$. While propositional satisfiability is NP-complete, propositional Horn clause satisfiability is linear (Dowling and Gallier 1984).

To derive results in logic programming a procedural reading of the above clause is used: "to prove P_1, try to prove P_2 and ... and P_n". Consequently in a logic program the above clause is written using the following syntax:

$$P_1 \leftarrow P_2 \wedge \ldots \wedge P_n$$

P_1 is termed the *head* of the clause.

The algorithm becomes more complicated when the program includes variables, i.e. we move from propositional logic to predicate logic. Suppose the program includes the clause

$$p(a) \leftarrow q(b, Z), r(Z)$$

This clause says that "to prove that the predicate p holds of the constant a, find some value for the variable Z such that $q(b, Z)$ and $r(Z)$ can be proven". Given the query

$$\leftarrow p(X)$$

which means "find some value for the variable X such that $p(X)$ can be proven", the logic programming system chooses the value a for X and uses the above program clause. The algorithm for finding candidate values for variables in an atomic query is by selecting a candidate clause, and *unifying* the query atom with the clause head[2]. The meaning of "unification" is simple. Two terms can be unified if there are values for the variables in the query which make the terms equal. Unification is, therefore, a (simple) way of handling equations. Indeed any logic program can be rewritten so that all the equations are explicit. For example instead of

$$p(a) \leftarrow q(b, Z), r(Z)$$

[2]The reader may notice that according to the definition in the following sentence, it is an abuse of language to talk about "unifying" a query atom and a clause head: only terms can be unified. However, our intended meaning should be clear, by viewing an atom as a term.

we can write

$$p(X) \leftarrow X = a, Y = b, q(Y, Z), r(Z)$$

The built-in, default, logic programming algorithm has two components. The first component is the algorithm that can, alone, handle propositional queries against propositional logic programs. The second component is the one needed to handle programs involving predicates and arguments: this is the algorithm that performs unification.

In the 1980's researchers began to investigate equations (DeGroot and Lindstrom 1986) and finally Jaffar and Lassez (1986) realized that equations between terms in logic programming could be replaced by inequations, or any other kind of constraint, provided certain conditions are satisfied.

CLP is simply the extension of logic programming to admit other constraints than syntactic equations. Like logic programming, CLP admits only Horn clauses. CLP programs are exactly like logic programs where all the equations are written explicitly (as above), except that now in place of equations are more general *constraints*. An example clause is

$$p(X) \leftarrow X > 1, Y = b, q(Y, Z), r(Z)$$

There is no once and for all fixed class of admissible constraints in CLP. Instead each particular CLP language has its own particular class of constraints. Hence the CLP Scheme is often expressed with a parameter *CLP(X)*, with the idea that CLP language is an instance of CLP(X) obtained by specifying a particular class of constraints for X. Naturally the specification of a class of constraints includes not only their syntax (i.e., which predicate and function symbols are admitted), but also their semantics. A class of constraints together with their semantics is often termed a *constraint domain*. Some example domains are linear constraints over the rational numbers $CLP(\mathcal{Q})$ and interval constraints.

In order to ensure correctness and completeness for CLP(X) an algorithm is required to decide for constraints over X whether they are satisfiable, in much the same way that unification decides syntactic equations. Such an algorithm is termed a constraint *solver*. The procedural reading of a CLP clause is similar to that of a logic program clause except for the handling of constraints. The above CLP clause reads "to prove $p(X)$ add the constraints $X > 1$ and $Y = b$ to the constraint store, test the consistency of the resulting constraint store, and prove $q(Y)$ and $r(Z)$". The result of the computation is the final constraint store[3]. For completeness, the only requirement on the solver is to be able to detect whenever the constraint store is inconsistent.

With regard to admissible constraint domains, two conditions must be satisfied. These conditions are satisfaction completeness and solution compactness. *Satisfaction completeness* requires that any set of constraints C is either satisfiable or inconsistent. This rules out constraint domains whose semantics are not expressive enough. *Solution compactness* ensures that the negation of a constraint can be approximated arbitrarily closely by other constraints.

[3]projected onto the variables appearing in the original query.

The idea of replacing syntactic equations with constraints in CLP has a natural parallel in databases, if we view a database tuple as a set of equations between variables and their values, as illustrated in the introduction. A generalized tuple is just a set of constraints over a given constraint domain.

The worst-case complexity of relational database querying rests on the assumption that an unquantified query can be answered in polynomial time. For this result to carry over to constraint databases, it is necessary that constraint solving itself has only polynomial complexity. Clearly the constraint solver is needed for each access to a generalized tuple, so very fast constraint solving is crucial. This is the background to one focus of current research on constraint databases (Kanellakis, Kuper, and Revesz 1995; Goldin and Kanellakis 1996; Brodsky, Goldin, and E.Segal 1996; Koubarakis 1996).

2.3 Concurrent Constraints

The CLP Scheme is very elegant, but it would never have made such an impact if there were not major industrial applications. Constraint programming has been particularly successful in factory scheduling, personnel rostering and transportation. The interested reader is referred to the proceedings of the conference on Practical Applications of Constraint Technology (The Practical Application Company 1996), the ILOG user group report (ILOG 1995), the CHIP user group report (COSYTEC 1994), and a recent survey (Wallace 1996).

The key to this success has been a move away from the idea of a black box constraint solver. First, any particular CLP language will typically include several constraint domains. Thus there will be several solvers each responsible for its own constraint store. Second, some of these solvers will be replaced by a ground constraint checker and a set of constraint agents.

A *ground constraint checker* for constraints $(\mathbf{X}|C)$ over variables $\mathbf{X}$, is able to check the consistency of the constraints once all the variables are instantiated. For example a constraint solver for linear constraints will detect the inconsistency of the pair of constraints $X > Y, Y > X$, however a constraint checker can only handle these constraints once X and Y are instantiated, e.g. $X = 1, Y = 3$. In this case, of course, the constraints can be checked independently, so constraint checking is much cheaper than constraint solving.

Constraint agents can handle constraints with uninstantiated variables, but do not guarantee to detect all inconsistencies. Constraint agents extract useful information from the constraints - for example upper and lower bounds on the variables. The information extracted is, in a sense, the *data* produced by the constraint agents. However a constraint agent is not like a procedure which executes once and then exits, but it is more like a process which receives inputs and generates outputs repeatedly during its life. An input might be a new bound for one of the variables involved in the constraint, and an output might be a tighter bound for another variable, which serves as input to another constraint agent. Thus constraint agents can cooperate in solving a set of constraints.

The data input and output from constraint agents, such as upper and lower bounds on variables, are themselves constraints. We distinguish them from constraint agents and call them *basic* constraints. Unlike constraint agents, basic constraints are not active: they are simply read and written (or "asked" and "told") to a *constraint store* (Saraswat 1993). A constraint store is not some physical computer memory, but a useful concept which helps in our understanding of constraint agent behaviour. Each constraint store is controlled by a constraint solver, which checks each new basic constraint when it is told to the store. The constraint solver reports a failure whenever an attempt is made to add an inconsistent basic constraint to the store. Because the constraint solver must enforce *global* consistency amongst all the basic constraints in the store, there is a potential efficiency problem. Therefore very simple classes of basic constraints are usually held in the store. For example upper and lower bounds on variables are very easy to check for global consistency: if the bounds on each variable are consistent (i.e. the upper bound is no smaller than the lower bound), then the constraint store is globally consistent. In this case global consistency is equivalent to local consistency.

A constraint store is best understood as an approximation of one current intermediate solution of the input query. Together with the constraint agents, the constraint store represents current the state of the computation.

Here is a simple example with four agents, working on the problem $3 \geq X, X \geq 1, X \geq Y, Y > X$. Each agent can extract information about upper and lower bounds. $A1$ handles the constraint $3 \geq X$, $A2$ handles $X \geq 1$, $A3$ handles $X \geq Y$ and $A4$ handles $Y > X$. These agents can work in any order, or all at the same time. Lets simulate their behavior assuming they all work in parallel. We write $\overline{X}$ for the upper bound of X and $\underline{X}$ for its lower bound.

Step	A1	A2	A3	A4
1	$\overline{X} = 3$	$\underline{X} = 1$		
2			$\overline{Y} = 3$	$\underline{Y} = 2$
3			$\underline{X} = 2$	$\overline{X} = 2$
4			$\overline{Y} = 2$	$\underline{Y} = 3$

The underlying finite domain handler reports at the end that the upper and lower bounds on Y are inconsistent and constraint handling stops. It is the facility of constraint agents to act *concurrently* that gives rise to the name "concurrent constraint programming" (Saraswat 1993). In the following we shall sometimes talk about the "behavior" of a constraint (agent). By this we mean how it acts in the presence of a given constraint store. For example the behavior of the agent $A3$ above, which handles the constraint $X \geq Y$, is to do nothing until the lower bound of Y becomes greater than the lower bound of X, or the upper bound of X becomes smaller than the upper bound of Y. At this point it tightens the appropriate bound on the other variable, and suspends itself again until another agent changes the bounds of X or Y causing it to wake up again.

Naturally these constraints could have been solved by a linear constraint solver, but the advantage of constraint agents is their efficiency and adaptability. For example as long a set set of linear constraints is consistent, these agents will typically do nothing, only "waking up" when an inconsistency arises. By contrast a linear solver acts globally on all the variables encountered so far, at every constraint check. It is possible to define constraint agents to handle specific kinds of constraints, such as those that arise in factory scheduling (Aggoun and Beldiceanu 1993).

These agents cooperate with any other agents that may be around, which makes the technology highly appropriate for industrial problems which typically involve a multitude of application-specific constraints in addition to the standard ones (Wallace 1995).

An important advantage of constraint agents of black box constraint solvers is that their behavior can be influenced by the behavior of the (constraint) logic program in which they are embedded. For example the order in which variables is controlled by the logic program, and it can have a dramatic influence on the behavior of the constraints agents (Van Hentenryck 1989a).

An implementation of concurrent constraints in a constraint database would require an extension to the representation of a query state - the current partial solution would no longer just be a temporary table, but a table where each row was augmented by a constraint store and a set of constraint agents. An open issue is how to avoid having multiple copies of the same agent - one for each tuple. Nevertheless the idea of adapting the concurrent constraints technology for database queries would appear to offer some important new optimization possibilities.

3 Application Areas

Both databases and constraint programming have been used in a wide range of industrial applications. In the introduction, we already highlighted some of the weaknesses of current current database systems and outlined how they could be overcome by using constraints. In this section, we investigate limitations of current constraint programming systems that could be alleviated by moving to constraint databases. Furthermore, we survey a variety of constraint domains, reporting on applications involving constraints of each kind.

3.1 From Constraint Programming to Constraint Databases

3.1.1 Scaling Up Combinatorial Problems?

Though constraint programming is still a fairly new discipline, it has found considerable attention in areas which involve notoriously difficult problems. Common to all these problems is that they belong to the class of NP-hard problems, i.e., there is no algorithm known that solves the given problem within polynomial time and space. Although the use of constraint programming does not cure this problem, it allows users to easily experiment with different problem representations.

Constraint logic programming is applied in many industrial applications in various areas. There are several reasons for this success: First, constraints are a suitable tool for representing intricate problems due to the high-level modeling they support. The semantically rich constraint programming languages provide a means to express complex problems in a concise way. Thus, the programs become quite readable and maintainable (Cohen 1990; Jaffar and Maher 1994). Second, related to the previous point, is that constraint logic programming offer support for rapid prototyping, that is, they support the swift encoding and changing of complex problems. Third, constraint logic programming systems typically allow users to control the search for solutions, which is often critical for the success of the search. Fourth, these systems can be coupled with standard mathematical programs such as linear program solvers. Lastly, techniques such as constraint propagation turned out to be a suitable tool for finding feasible solutions efficiently. All these reasons contributed to the success of constraint logic programming, which is reflected by the wide range of applications (Wallace 1996) they are used for.

However, from a database point of view there is caveat. Many of the problems investigated by the constraint logic programming community operate on a small amount of data. Even though these intricate problems crop up from time to time, we are faced in our daily lives with a reverse setting: few constraints and a large amount of data. For instance, suppose you have to organize a trip which takes you to four cities. This trip involves not only booking the flights, but also making the hotel reservations, organizing tickets for the train and so on. As constraints for the trip, you impose that you do not want to wait longer than one hour and that the price should be less than a certain upper bound. Furthermore, you want to fly with a certain airline. Constraints are quite convenient to express this kind of query and to solve the underlying scheduling problem. Clearly, in the framework of constraint databases such a query could be answered efficiently, since all the data relevant for the search is already expressed as constraints. By storing intermediate results along with their constraints, it is possible to perform rescheduling without having to start from scratch.

3.1.2 Loose and Tight Database Couplings

In constraint logic programming, as in logic programming, there is no distinction between programs and data. One consequence is that for large applications, involving a lot of data, programs can become very large. Even though plummeting cost of main memory makes it often possible to run very large programs quite efficiently without having to store parts on secondary memory, there is still a need for database support. For example, in large applications involving several users concurrent access to data is very important. Furthermore, persistence as offered by databases is crucial when programs have to run repetitively for different parameter settings or when intermediate results have to be stored for further use.

One way of providing database support for constraint logic programming is to use a *loose* coupling to an external database. This approach has been successfully applied for industrial applications (Kay and Simonis 1995), but it incurs severe runtime overheads in case repeated database accesses are required throughout program execution. Even though loading a superset of the required data from the database into the constraint logic programming system at start mitigates this problem, it does not solve it in a satisfactory way. A second disadvantage of this approach is that the constraints cannot be stored in the database in a form that reflects their meaning. For example, they may be stored as strings. As a result, after having been retrieved from the database, the constraints must be converted from the database representation into a form which can be used by a constraint logic programming system. Consequently, it is not possible to handle constraints held in the database with the same efficient algorithms that are used for handling constraints in the main memory constraint logic program.

A constraint database, on the other hand, could support a *tight* connection between a constraint logic programming system and a database. For logic programming an example of a tight connection is the ECLiPSe/Megalog system (Bocca 1992). The issues to be tackled in building a tight connection between constraint logic programming and a database are

- Transactions
- Optimization
- Indexing

In a loose connection, the first issue is handled for free by the database management system.

The issues of optimization and indexing are wide open. For relational databases efficient query evaluation has always relied on a set-at-a-time evaluation to minimize the number of pages retrieved from disk. For object databases this approach has come into question because the set-at-a-time approach is often not applicable and it is not clear at this moment whether the set-at-a-time paradigm can be fully carried over to constraint database systems.

Constraint logic programming systems find solutions one-at-a-time. Let us consider, in particular concurrent constraint programming systems (see Section 2.3). These systems use constraint agents acting concurrently on a constraint store. The constraint store approximates a single intermediate result, not a set of results. The behavior of each constraint agent is driven by the contents of the constraint store. This behavior in turn adds more information to the store. Any adaption of such a system to a set-at-a-time evaluation, would have to deal with the problem that the different stores - approximating different intermediate results - may hold a different amount of information. If the different intermediate results were held in an intermediate relation, for example, then different tuples may have different attributes, reflecting the different information accumulated about each constraint store by the constraint agents.

Such flexible query evaluation behavior is not possible set-at-a-time. The payoff between set-at-a-time and tuple-at-a-time evaluation for constraint databases is not clear and certainly depends upon the application and the constraints that appear in it. In the DISCO constraint database system (Byon and Revesz 1995), a set-at-a-time approach has been chosen, though it is designed for NP-hard queries. This contrasts with the usual technique for solving hard problems in constraint logic programming, which uses constraint propagation specifically to guide future choices and restrict search.

3.1.3 Persistent Constraint Stores

The concept of a constraint store was introduced in Section 2.3. This concept extends the traditional notion of a computer program store in two ways. First, it allows partial information to be stored: for example if the constraint store keeps the upper and lower bounds for a variable, then it admits any value between the two bounds as a possible value for that variable. For example, it might record that a person's age lies between 50 and 60. Second, it supports specialized built-in consistency checking. Thus if the person was also known to be older than someone else, whose age was 65, an inconsistency would be detected.

The handling of partial information is not a source of theoretical or practical problems within the constraint logic programming framework. Suppose constraint stores could be made persistent: would this offer a new and effective approach to the handling of unknown values in databases? The possibility is intriguing, but it brings up a variety of issues which are beyond the scope of this introduction. Suffice to say that in the constraint logic programming framework, constraint stores are not required to offer the full range of queries that must be handled by a database system. Nevertheless the embedded facility to check consistency and entailment between constraints of certain well-defined and useful classes provides excellent support for solving hard combinatorial problems of the kind introduced before.

In the remainder of this section, we shall discuss some of the most important constraint domains and applications in which there are used.

3.2 Constraint on the Real Numbers

All quantities which can be repeatedly subdivided are representable using real numbers. Obvious examples are length, width, height, and generally distance, area and volume. Time is also divisible into smaller and smaller units, if we allow fractions of a second, so time too can be represented on a real number scale. Combining time and distance we can represent trajectories and journeys, flows, and the movement of points, areas and volumes in a machine under design. Other examples of continuous quantities are temperature and weight, and electrical current and resistance.

Constraints on these quantities are therefore expressible as real number constraints. In general a constraint over real numbers may be any predicate calculus formula built from equality and inequality between real number expressions. These expressions may involve arithmetic functions, trigonometric functions, rational and irrational constants. There is no general algorithm for solving all real number constraints, though there are some very powerful algorithms which can solve systems of polynomials (Buchberger 1985; Monfroy 1992) and quantified formulae involving polynomials (Tarski 1951; Collins and Hong 1991).

For constraint databases, where the number of constraints to be solved may grow large, these algorithms have to be scrutinized regarding their applicability in a database context. The major problem for their immediate application is that their computing requirements grow very fast with the number of variables and constraints. However the application of constraint agents operating on a store of interval constraints (Van Hentenryck, McAllester, and Kapur 1995), and of hybrid numeric solvers (Benhamou and Granvilliers 1996), offer some promise for handling real numeric constraints which can function in a database context. Moreover, the number of constraints to be solved within in a single database query is unlikely to grow with the size of the database: especially as constraints in separate answer tuples can be handled separately.

Nevertheless when the number of tuples accessed during query evaluation grows large, then constraint handling for each individual tuple must be highly optimized. Therefore, techniques such as interval reasoning (i.e., reasoning about the maximum and minimum possible value for the variables) are very interesting in the context of constraint database systems, even though they are much weaker than constraint solving

In the subsequent sections, we show that constraints over the real numbers are very important for a multitude of application areas such as *Engineering and Science*, *Medicine* and *GIS and Environmental Information Systems*. These three areas have been identified in a recent workshop on Visual Information Management Systems (Jain 1993) as key applications.

3.2.1 Engineering

In Engineering the main applications are computer aided design, manufacture of the final artifact, and ongoing support and development. An extreme example is an aircraft, which contains hundreds of millions of parts. Clearly the maintenance database for such an object is huge and complex, and aircraft maintenance information management is currently supported by specialized languages and systems. However even photocopiers are extremely complex pieces of apparatus, and problems arise not only in the design of the different modules, but in attaching them (eg a particular paper feed device needs a tailored control behavior when it is attached to a particular photocopier) (Fromherz and Saraswat 1995).

An important need that will be served by a constraint database is bringing together different kinds of information in a single query. For the maintenance of buildings, engines and all kinds of plant, there is the need to combine spatial and alphanumeric information. Using a CAD system it is possible to view an object from any angle. In a combined system it would be possible to ask for a view which reveals a certain component - or all components with certain properties specified in the query. A more complex request would be to show the object with enough pieces removed so as to reveal the requested component.

Supporting the design process requires an active information system, since it must also give feedback on design errors. An obvious error is that two solids overlap in space/time. Although these kinds of errors are often detected during simulations, constraint database systems could immensely contribute in avoiding these and more subtle errors. By specifying integrity constraints many such errors can be picked up immediately by the system without the need for the designer to run a possibly incomplete set of simulations. Other kinds of errors could also be detected as constraint violations: for example a wire which cannot carry the electrical current specified in the design (Shankar, Gilbert, and Jampel 1996). All these requirements are expressible as constraints over the real numbers, and can be handled in a constraint database by the appropriate combination of cooperating constraint solvers.

3.2.2 Medicine

Medicine is an area of highly complex decision making where there is already a large amount of information held on computers, but this information is not integrated. A long term application is to answer queries about a patients response to treatment using both numerical information (e.g., temperature and liquid consumption) and visual information (change in color of injured region, reduction in size of swelling, etc.). The advantages of using constrains in this application area are manifold. For instance, if a person is known to have an allergic reaction to some drugs, then this could be expressed conveniently in terms of constraints. And at the start of each medical treatment, the constraint database system could be consulted to see whether a certain treatment does not violate any known constraints.

As to shorter term problems, constraints are already being applied in bio-informatics. The problem of determining the order in which certain components appear in the genome is a crucial one studied by scientists all over the world. Clark et al. (1993) and Doursenot et al. (1993) describe how constraints are used for finding highly likely sequences.

Another combinatorial application is to predict the three dimensional shape of proteins. Large quantities of data on genome and on protein structures are stored in specialized databases. The Imperial Cancer Research Fund have already started to use a deductive database (ECLiPSe/Megalog) to research in this field and a move to a constraint database would follow naturally.

As research into this intricate area shows, these problems are too complex for a precise solution over the real numbers. So the protein folding problem is simplified by allowing only a small set of possible positions for each unit. The problem is then reduced to a combinatorial problem.

Other areas in which constraint database have a great application potential concern resource allocation - for example nurse scheduling (Heus and Weil 1996) - but this is the subject of a later section.

3.2.3 GIS and Environmental Systems

Spatial databases have been identified as one of the most challenging applications for database systems, because of their unique requirements. A spatial database system must provide a flexible and extensible data model, complex user-defined data types and predicates, multidimensional indexing methods, and so on.

Even though relational database systems are quite efficient for handling textual information, they fall apart when it comes to storing geographic information. The fixed structure as imposed by relations is unsuitable for storing polygons with a varying number of vertices without facing major performance penalties while retrieving the data from secondary storage. Object-oriented or extend relational database started out with the promise to overcome these drawbacks by offering a more powerful data model. However, these systems are far from being the ultimate solution. Besides their inability to optimize queries containing user-defined functions, there is virtually no support for multidimensional indexing methods (Gaede and Riekert 1996).

Constraint database systems, on the other hand, have the potential to overcome some of the problem discussed before. For example, constraints allow the representation of sets of points and polygons in a quite natural way. Furthermore, constraints can be exploited for conventional data processing as well as for the evaluation of spatial query predicates like intersection and distance. Related to the last point is that constraints lead to a concise and natural definition of spatial predicates that nicely integrates into declarative constraint query languages. Lastly, by considering multidimensional indexing techniques as constraints it is further possible to optimize, or better, to constraint the search for relevant data.

Geographic information systems have found considerable attention in the past. Commercially, it seems that particularly Utility Companies and Marketing people are among the most interested users groups. *Utility companies* have a complex physical network which needs to be designed, extended, modified in the light of events, and maintained. The use of GIS in *marketing* is increasing rapidly, as it provides very valuable information for targeting marketing effort and sales outlets. Textual, numeric, and especially aggregate information are superimposed on a map so that areas with a high density of likely customers can be easily identified. This kind of application needs to be linked to other decision support applications such as warehousing and transportation, where constraint logic programming is already being successfully applied. Again, constraint database system would contribute to facilitate some of the tasks in these areas.

Routing is a typical GIS problem and there are probably infinite variations depending on the constraints of the application. In planning and operating a mine, escape routes must be planned for each point in the (growing) network of mine shafts and each of the escape routes must naturally satisfy specific safety constraints (Burgard, Cremers, Kolbe, and Plümer 1996). Note that this application exemplifies a three dimensional geographic information system.

Vehicle routing is an application area in itself. A number of industrial constraint logic programming applications address the problem of how to use a fleet of vehicles most efficiently to satisfy customer demand (Kay and Simonis 1995). Typically the relevant point-to-point distances are precoded, thus excluding the need for an underlying GIS system. FISSURE addresses route finding in a continuous domain (Yakhno, Cheblakov, and Zilberfain 1995). Closely linked to the transportation problem are the related problems of where to place warehouses and shops so as to minimize transportation costs and maximize delivery targets. An industrial example implemented in constraint logic programming is the EBI system (Simonis 1995).

A weakness of traditional GIS systems is the lack of support for time. Applications involving GIS and temporal reasoning include navigation and maneuvering (Benjamin, Viana, Corbett, and Silva 1993), sensor management (Aschenbrenner, Brodsky, and Kornatsky 1995), local government and real estate conveyancing (Puddicombe 1996), waste disposal, environmental management and protection (Müller 1995). Hermosilla (1993) gives a survey on this topic. Roman (1990) specifies the requirements for supporting space, time and accuracy in a database system.

3.2.4 Temporal Databases

The storage and retrieval of time related data in databases has found considerable attention in recent years, is still an active field of research. One of the first insights of this research was that conventional database (e.g., relational) and their query languages only poorly support time. As a result, there have been proposals on how to change existing database models and query language, to cope with the special requirements of time, but none of these temporal database models represents a unified framework. In particular, they do not integrate well *definite*, *indefinite*, *finite*, and *infinite* temporal information (Koubarakis 1993). On the other hand, the combination of the theory of relational database systems, *temporal constraints* and *conditional tables* to *temporal constraint databases* offers an integrated framework, capable of dealing with indefinite temporal information. Furthermore, this database model allows reasoning over time by means of temporal constraints.

Constraints have also been identified as a natural means for the specification of similarity queries as they occur for example in databases storing time series data (Goldin and Kanellakis 1995). This kind of data can often be found in areas such as physics or medicine or in financial applications, where a process is observed for a certain time and at equidistant time points a measurement is conducted. Thus, time related data is typically represented as a sequence of real numbers, possibly associated with some uncertainty bound. In other words, the measurement represents an *approximation* of the exact value. An important concept of time series data is similarity, which is typically defined by means of some application dependent distance metric and some ϵ-bound. Two time series are considered as being similar if their distance if less than the given ϵ-bound (Jagadish 1991; Jagadish and Mendelzon 1995). Typical queries in a time series database are: "Given a time sequence of a certain length, find a sequence stored in the database that is similar to the given one" or "given a time series, is there any sequence in the database which is shifted by a certain amount relative to the given one" or "Find pairs of similar sequences in the database". As it turns out, constraint database systems along with their query language offer an ideal tool for answering and expressing these kind of queries (Goldin and Kanellakis 1995).

3.2.5 Financial Systems

For reasoning about investment and income it is often necessary to solve non-linear problems. However, it is possible to calculate compound interest by iteratively computing simple interest over short time periods. This is the approach typically used to tackle financial applications using constraint programming. It stands out as an example where constraints combine intimately with recursion. Two industrial examples solved using constraint programming are an expert decision-support system for option-based investment (Huynh and Lassez 1990) and the SEVE system (Simonis 1995). The database requirement in applications of this sort are probably not because of the data quantity, but more because the system must use up to date information which is held in an external database.

Spreadsheets are a very powerful computing paradigm and the integration of constraints adds a great deal of flexibility and power. Hyvőnen (1991) discusses the application of constraint-based spreadsheets in the area of finance.

3.3 Finite Domain Constraints

Finite domain constraints are closely related to interval constraints, in that each constraint specifies a set of possible values for a variable - but in this case the number of possible values is finite. By themselves finite domain constraints are too weak to express any non-trivial problem: real problems have a multitude of other constraints in addition. In a *constraint satisfaction problem* (CSP), all the variables have a finite domain constraint, as well as other constraints. There is in principle a simple algorithm for solving CSP's: test each combination of possible values for the variables until one is found that satisfies all the problem constraints. There is a long history of research in Artificial Intelligence to develop ways of improving on this naive algorithm (Montanari 1974; Freuder 1978). Van Hentenryck (1989b) shows how to incorporate some of these techniques into logic programming.

The most important technique for constraint satisfaction problems is to use other constraints to remove impossible values from the finite domains of the variables. There are standard algorithms for this (Mohr and Henderson 1986), but for industrial problems such as scheduling, personnel assignment and transportation specialized constraints and constraint behaviors (see Page 12) have been developed. The specialized behaviors have the effect of removing impossible values from the finite domains of their variables. Doing so has the great advantage that different constraints can act independently, communicating with each other solely through the finite domains of their common variables.

For scheduling a specialized behavior has been designed (Caseau and Laburthe 1994), which is based on an algorithm developed in the area of operations research (Carlier and Pinson 1994). Simonis and Cornelissens (1995) and Puget (1995) report how technical achievements have been applied in industry.

For personnel assignment another specialized constraint has been developed which enforces specified constraints over a rolling time window (e.g., "any on duty period lasting more than eight hours must be followed by at least 12 hours off duty"). Examples of personnel scheduling include (Lazaro and Aristondo 1996; Feldman et al. 1995).

For transportation it is important to build circular routes for which a special *cycle* constraint has been defined.

Finite domain constraints can be used for modeling and solving continuous problems by splitting the continuous domain into a finite number of small pieces. Thus a class of problems, such as routing problems, has instances which are handled by continuous constraints, and other instances solved by finite domain constraints. There are many problems which involve both continuous and finite domain constraints. This class of problems is well known in the field of operation research and is called the class of Mixed Integer Programming problems (MIP). Interfacing of MIP solvers to databases - both the semantic issues and the implementation issues - is an important development in operation research field just as the development of constraint databases is important for constraint programmers. These different approaches are quite complementary and alliances are being formed between constraint logic programming and MIP groups.

3.4 Boolean Constraints

It was recognized quite early that Boolean constraints are highly appropriate for integration into the CLP(X) framework (Büettner and Simonis 1987). The reason is that syntactic unification, as embedded in logic programming, has a precise counterpart for Boolean constraints, namely Boolean unification. It is possible to extract from a set of Boolean constraints the most general constraints on the variables that they entail. These most general constraints are equivalent to the input Boolean constraints and serve as a canonical representation that is highly suitable for logic programming.

The availability of a most general solution is very useful for applications with many alternative possibilities. For example it is extremely hard to establish the correctness of an integrated circuit, because there are so many possible input combinations to test. A Boolean variable represents each output of the circuit and another to represent each intended output. Finally a single result variable can be used represent the correctness of the circuit by constraining it to take the value true if and only if the outputs are the same as the intended outputs. The actual, and intended, circuits are captured in terms of Boolean constraints and the whole set of constraints is passed to the Boolean solver (Bormann, Lohse, Payer, and Schmid 1995).

The same principle applies for proving properties about finite state machines, and makes it possible to verify properties of real-time control software (Fromherz and Saraswat 1995).

Boolean constraints have also been used for continuous domains: where conjunction maps to set intersection and disjunction to set union. This offers an approach to the optimization of spatial queries (Helm, Marriott, and Odersky 1995).

3.5 Set Constraints

One application of set constraints is for reasoning about programs (Heintze and Jaffar 1994). However, set constraints are also used for handling combinatorial problems. For this purpose a rather specific set representation has been developed, which associates an upper and lower bound with each set variable. A technique similar to those used for finite domain reasoning can be applied to narrow the bounds during problem solving. The lower bound is increased by inserting elements that *must* belong to the set, and the upper bound is reduced by removing elements that *cannot* belong to the set (Gervet 1994; Revesz 1995; Puget 1996).

Set constraints offer a solution to the problem of *symmetries* which frequently cause severe performance problems in the solving of combinatorial problems. Examples of their use include airline crew scheduling (Pavlopoulou et al. 1996), maintenance crew scheduling (Feldman et al. 1995) and a configuration problem (Mailharro and Le Quenven 1995). They have also been used to model genome mapping and sequencing problems (Revesz 1996).

All these examples could also have been modeled using finite domain constraints, but set constraints provide a more natural modeling language as well as dramatically reducing the symmetries in the problem's search space.

3.6 Dynamically Changing Constraints

Burgard, Cremers, Kolbe, and Plümer (1996) present a mine planning application which also illustrates the need for a tight connection between the database and the end user interface as needed for interactive development of solutions to complex problems. The maintenance of an up-to-date image of a database object in a window can be treated in precisely the same way as other constraints on the object. Indeed this is the normal method of user-interface support in constraint programming applications. A commercial constraint library for constructing interactive graphical user interfaces is presented by Borning and Freeman-Benson (1995). Object Technology International has used the library to build a graphical editor for a large database.

Constraint-based graphics appears to offer a very clean, powerful way to express graphical interfaces, and graphical languages (Helm, Huynh, Lassez, and Marriott 1992). As yet the technology has not scaled up sufficiently to allow such general constraints to be tackled in large graphical applications such as commercial CAD systems (but see (Mezzanatto, Foglino, Giordanengo, Apra, and Gullane 1993) for a specialized application, and (Gleicher 1994; Donikian and Hegron 1994) for ongoing developments).

One very important industrial problem is what to do when things don't go as planned. This is very obvious in the travel industry. When a plane is delayed, or a train breaks down, the whole timetable is affected. In fact any industry using its resources efficiency is liable to knock on effects when something goes wrong. Constraint databases could be used to replan after an unexpected event. One small example is the application of constraint programming techniques to a workforce scheduling problem where new tasks are continually arriving, and actual durations of tasks vary from those planned (Azarmi and Abdul-Hameed 1995).

Even more dynamic is the control of robots, and other automated systems. Given a constraint-based specification and a model of the plant and the environment, automatic synthesis of a correct constraint-satisfying controller becomes feasible, as shown for a simple ball-chasing robot by Zhang and Mackworth (1995).

Gupta, Jagadeesan, Saraswat, and Bobrow (1995) recently developed another approach for modeling timed *reactive systems*. Reactive systems are those that react continuously with their environment at a rate controlled by the environment. Execution in a reactive system proceeds in bursts of activity. In each phase, the environment stimulates the system with an input, obtains a response in bounded time, and may then be inactive (with respect to the system) for an arbitrary period of time before initiating the next burst.

These ideas may support both a formalization and an implementation approach for active databases.

4 Theoretical and Practical Issues

In this section, we present some of the most recent results of research into constraints databases. A number of these results are theoretical in nature, so we start by introducing some of the fundamental concepts on which they are based.

4.1 Database Queries and Complexity

4.1.1 Expressive Power and Data Complexity

The two competing requirements of a database query language are high expressive power and low query evaluation cost. Naturally it is easy to design a Turing complete query language, but there are a variety of ways of limiting the expressive power in order to reduce the worst-case query cost.

In attempting to formalize results about expressive power, there are two "handles" we can get on a language or pair of languages. Firstly we can identify a certain difficult database query, and explore whether this query can be expressed in a given query language. Secondly we can establish that one query language is more expressive than another by showing it can at least express every query expressible in the second language.

In attempting to formalize results about query evaluation cost, we use complexity theory. For each query language we devise an algorithm to evaluate queries expressed in the language and count the steps required when evaluating the algorithm on a theoretical machine. Obviously this number depends on the size of the query and on the size of the database. In the database context, the size of the database is the most important factor, because our requirement is that queries should still produce an answer in a reasonable time even when the amount of data is enormous. Also the influence of the size of the query result is factored out by measuring the cost of determining if a given data object is in the result. Thus the measure used is *data complexity* which measures how the worst case cost of a yes/no query grows with the size of the data in the database.

4.1.2 Query Languages

Because of its simple logical basis, database theoreticians have used the relational data model in formalizing queries. In reasoning about constraint databases they use the extension of the relational model to the generalized relational model. Specifically a *generalized relation* is a finite set of *generalized tuples*, of the appropriate arity. A database comprises a finite set of generalized relations.

Two classes of query languages are considered:

- **Relational Calculus:** *RC*. This is a query language in which the only predicates and variables involved in the query evaluation are those explicitly mentioned in the query. In particular the language cannot express recursive queries.

- **Datalog:** *Dlog*. In order to strictly contain the relational calculus, the actual language used is *inflationary Datalog*$^{\neg}$. A Datalog query is, in fact, a program which may include recursion. In effect the cost of evaluating the query is not bounded by the number of procedures in the program, but by the number of different answer tuples it could possibly produce.

In terms of expressive power, relational calculus is equivalent to recursion-free Datalog$^{\neg}$: the Datalog program can be unfolded into a single query at compile time. However the unfolding can produce an exponential increase in the size of the query.

In the subsequent sections we will encounter a variety of constraint domains. Some important constraint domains explored by the theorists include

- **Syntactic Equality** Constraints (EQ) are equations of the form $x = y$, $x \neq y$, $x = c$, $x \neq c$ where x, y are variables and c is a constant. The underlying domain is infinite.

  Example: $(x_1, y_1) = (1.4, 2.5)$

- **Dense Linear Order** Constraints (DLO) are equations and inequations of the form $x = y$, $x > y$, $x < y$, $x = c$, $x > c$, $x < c$, where x is a variable and c a constant. The underlying domain is an infinite set on which an irreflexive, dense linear order, which interprets the $<$ symbol is defined.

  Example: $1.58983 \leq x_1$

- **Polynomials** Constraints ($Poly$) are polynomial equations and inequations, $P = 0$, $P \geq 0$, $P \leq 0$, where P is a polynomial with real coefficients. The underlying domain is the real numbers.

  Example: $1.4x_1 + 2.3x_1 \cdot x_2 + 0.9x_3 < 5$

Each of these query language classes can be instantiated to a specific query language by choosing a constraint domain - exactly as the Constraint Logic Programming Scheme, *CLP(X)*, is instantiated to a specific constraint programming language, by instantiating X. Thus we encounter *relational calculus over polynomial constraints*, *RC(Poly)*, and *Datalog*$^{\neg}$ *over dense linear order constraints*, *Dlog(DLO)*, for example.

It is a standard requirement on database query languages that the result of any query must be representable in the database. This requirement is called *closure* and is essential for databases. In the particular case of constraint databases the result of a query must be a generalized relation. The requirement for closure in the presence of negation imposes a stronger condition than solution compactness (see Page 10). We require the negation of a generalized tuple to be expressible as a *finite* disjunction of generalized tuples. Different instances of *RC(X)* and of *Dlog(X)* are closed. For example *RC(Poly)* is closed but *Dlog(Poly)* is not (Kanellakis, Kuper, and Revesz 1995).

4.1.3 Complexity Classes

In concluding this introduction we summarize some basic complexity classes, which are important for database query languages. The first complexity class is *PTIME* - problems which can be computed on a deterministic Turing machine in a time polynomial in the size of a given parameter (keeping the other parameters constant). In studying database query language, the

data complexity	Machine	Meaning
PTIME	Turing	uses polynomial time
LOGSPACE	Turing	logarithmic space on one tape
NC	Parallel Random Access	polynomial no. of processors running in polylogarithmic parallel time
AC^0	Random Access Alternating	with a fixed number alternation running in logarithmic time

Table 1: Taxonomy of Data Complexities

parameter we allow to vary is the size of the database; the size of the query is kept constant.

A cheaper complexity class is *LOGSPACE* - problems which can be solved using an extra space, over and above the space needed for input and output, which increases logarithmically with the given parameter. *LOGSPACE* languages are a subset of the *PTIME* languages.

Another cheaper class is *NC* - problems which can be solved efficiently on parallel machines. *NC* is also a subset of *PTIME*. A special class of *PTIME* languages are those that can express any polynomial algorithm - the *P-complete* languages. It is highly likely there are queries expressible in *P-complete* languages that do not belong to *NC*.

Finally the class AC^0 is the first member of an increasing sequence of class $AC^i : i \in \mathcal{N}$ whose union is *NC*. Thus AC^0 is, trivially, a subclass of *NC*. Table 1 gives the interpretations of the most important data complexities.

In the remainder of this section, we shall first highlight some recent data complexity results before turning our attention towards aspects related to the expressiveness of query languages.

4.2 Query Complexity

The syntax of the constraint query language by Kanellakis, Kuper, and Revesz (1995) is defined by combining a database query language (relational calculus or inflationary Datalog$^{\neg}$) with some decidable logical theory (i.e., constraint domain). The semantics is based on the decidable theory. As it turns out, most of these combinations have low data complexity, i.e., each query can be evaluated efficiently with respect to the database size (see Table 2 for a summary of the results). The underlying execution model is as follows. First, the query is unfolded by substituting query symbols by their respective definition. Second, quantifier elimination is performed. The second step can be done in PTIME efficiently. Within this framework, they showed that finite relations can be generalized to finitely representable relations. Or to put it differently, to represent a potentially infinite number of tuples using a finite number of constraints. This framework also incorporates previous results in the areas of extended relational algebras, temporal data and the like. Further it demonstrates that constraints integrate nicely into current database systems and that relational database queries can be regarded as limited form of constraint programming.

Query Language	Constraint Domain		
	Polynomial	Dense Order	Equality
Relational Calculus	NC	LOGSPACE	LOGSPACE
Datalog$^{\neg}$	Not closed	PTIME	PTIME

Table 2: Data complexities of constraint query languages which are the union of some query language and a constraint domain according to Kanellakis et al. (1995).

In a later paper, Kanellakis and Goldin (1994) present an algebra for constraint query languages. By slightly modifying the constraint domain from dense order to dense linear order, the authors strengthened previous results. The result are summarized in Table 3, where =PTIME denotes that this combination exactly expresses PTIME. A similar proposal slightly biased towards spatial databases was presented later by Belussi et al. (1995).

Query Language	Constraint Domain		
	Polynomial	Dense Linear Order	Equality
Relational Calculus	NC	AC^0	AC^0
Datalog$^{\neg}$	Not closed	= PTIME	PTIME

Table 3: Data complexities according to Kanellakis and Goldin (1994).

Despite its conceptual elegance the framework of Kanellakis et al. does face some limitations (Brodsky, Jaffar, and Maher 1993; Kuper 1993; Grumbach and Su 1996). First, returning a quantifier-free conjunction as query result is hardly sufficient, since one is often interested in a concrete value instead of system of equation describing the solution. Second, aggregate functions (e.g., area, min, etc.) cannot easily be integrated into their constraint query language framework. These and other issues are addressed in the following section.

4.3 Expressive Power

4.3.1 Infinite Queries

In a traditional database only a finite number of data items can be stored. Therefore the truth of a universally quantified query (i.e. "Is it true for *all* the - infinitely many possible - data items X ...") will typically depend on data items which do not belong to the database! Those queries which only depend on data items stored in the database are called *domain independent*

(Topor 1987). Relational query languages impose a syntactic condition called *allowedness* which ensures queries are domain independent (Codd 1972; Van Gelder and Topor 1987). This condition ensures that any variable appearing in the query ranges over data items stored in the database. Quantified variables in allowed queries typically appear in the context $\forall X.R(X) \rightarrow \ldots$ or $\exists X.R(X) \wedge \ldots$, where R is some condition that only holds of stored data items. Even these syntactically restricted classes are powerful enough to express all domain independent queries (Codd 1972; Cooper 1982).

However a constraint database can store an infinite number of values - for example all the points on a line, or all the points on a plane - so there are universally quantified queries $\forall X \ldots$ which are true even though X ranges over an infinite set of points. It is therefore appropriate to relax the *allowedness* condition, but this raises some new questions about the expressive power of query languages.

One example query asks "Is a given constraint relation contained in a line?". As a non-linear query (whose constraints are polynomials, see Page 26 above) this can be expressed as $\exists A, B, C.\forall X, Y.(r(X, Y) -> A.X + B.Y = C)$ where r is the stored constraint relation. It turns out that this query cannot be expressed in a linear constraint database, where generalized tuples and queries employ only linear constraints (Afrati, Cosmadakis, Grumbach, and Kuper 1994). This example also shows that non-linear queries and databases are more expressive than linear ones, which is to be expected.

4.3.2 Generic Queries

A more subtle question is whether constraints add to the expressive power of languages for posing *queries which do not involve special constraint functions and predicates*. For example can some constraint domain X enable recursive queries to be expressed in the relational calculus language *RC(X)*? This issue is best understood using a concept called *genericity* (which is a generalization of domain independence).

A query is *totally generic* if it gives the same answer regardless of possible changes of the names of the data items. More subtly, a query is *monotone generic* with respect to an ordered constraint domain if it gives the same answer regardless of possible changes to the names of the data items, as long as their ordering is preserved. Thus if a and b are names, and $a > b$ held before the change, then $a > b$ still holds afterwards. Further kinds of genericity have been introduced in the literature, but we shall not discuss them here.

The transitive closure query is totally generic and, *a fortiori* monotone generic. Benedikt et al. (1996) show that the same monotone generic queries[4] are expressible in *RC(EQ)* as in *RC(Poly)*: in other words adding mathematical functions and relations to the query language does not allow the transitive closure query to be expressed. This result is proven for relational

[4]The result is actually proved for *locally generic* but these are equivalent to monotone generic under certain "mild restrictions on the order".

databases. However, Stolboushkin and Taislin (1996) lift the same result to constraint databases allowing both linear and polynomial constraints.

In fact the addition of mathematical functions and predicates adds no expressive power, for monotone generic queries over constraint databases (where the constraint domain is the real numbers). However it may offer a compact way of expressing generic queries - no polynomial size translation from *RC(Poly)* to *RC(Eq)* is possible (Stolboushkin and Taislin 1996).

4.3.3 Aggregate Queries

The consequences of adding aggregations to constraint databases has been studied to some extent by Kuper (1993). His major observations are that aggregate functions in constraint database systems have to take more arguments than their relational counterparts and that aggregate operations tend to produce results that belong to another constraint domain, i.e., they are not closed. Kuper (1993) suggests several ways to mitigate this problem such as reducing the number of aggregates allowed, restricting the query language, or using a typed language. In a subsequent paper, following the first of the above approaches, Chomicki and Kuper (1995) introduce a generic aggregation operator which returns for a given set how much data space it covers. They further show that relational algebra augmented with this operator satisfies the closeness property. Although their work can be used for evaluating aggregate functions like area, they cannot compute aggregates like perimeter. By requiring that the set of variables upon which is aggregated is independent from remaining the variables, Chomicki, Goldin, and Kuper (1996) define a restricted version of aggregation. As it turns out, the addition of the so restricted aggregation to relational algebra results into a closed, but restricted, constraint query language.

Grumbach and Su (1996) tackle the above problems from different point of view. Using a finite precision semantics (i.e., admitting only the finitely many numbers representable on a computer, instead of all the real numbers), they show how to integrate and evaluate aggregate functions. As a result of the finite precision semantics, it is no more possible to use classical quantifier elimination. Furthermore, the results of an arithmetic computation depends on the order in which subexpressions are evaluated (e.g., the distributive law does not hold anymore). To cure this problem, the authors propose a quantifier elimination algorithm allowing for the peculiarities of this semantics. Another result of their study is that query language enhanced with aggregates as well as Inflationary Datalog$^{\neg}$ with finite precision semantics can be still evaluated in PTIME.

4.4 Indexing

In order to be viable solution for storing and retrieving large sets of constraints, constraint database systems must be I/O efficient. In relational database systems I/O efficiency is accomplished by index structures such B-trees, which allow the database system to perform a well-aimed search for relevant data. Thus, the index enables the system to disregard most of the data stored. Although relational database systems are inherently multidimensional, most existing systems offer only one-dimensional index structures. This is particularly sad, because the spatial database community has designed in the past twenty years more than fifty multidimensional access structure (Gaede and Günther 1995b). Common to almost all these structures is that the indexed objects have to be approximated by simpler shapes such a minimum bounding boxes. These boxes together with a reference to the objects they approximate, are inserted into a spatial index. Due to the approximation, query processing is done in two steps: a filter and a refinement step. During the *filter step*, the spatial index is visited so as to identify objects that may satisfy the initial query. In the subsequent *refinement step*, the shape of the corresponding objects are loaded from secondary storage to test whether they really satisfy the query.

Even though for none of these access methods exist worst case boundaries for searching, updating, and storage utilization, they demonstrated in numerous empirical test with weird data distributions that the average performance is quite good. Thus, they are good candidates for indexing constraints.

In particular, access methods capable of storing polyhedra like P-trees (Schiwietz 1993; Jagadish 1990) or the cell tree (Günther 1989) are good candidates, to be used for indexing in constraint database systems. As recent empirical tests demonstrate (Günther and Gaede 1996; Goldstein, Ramakrishnan, and Yu 1996) these structures are likely to outperform rectangular-based access methods, since intervals are in contrast to common belief (Kanellakis et al. 1993; Brodsky and Kornatzky 1995; Brodsky et al. 1995) do not always represent a good approximation of polyhedra. This has also been recognized in a recent paper by Brodsky, Lassez, Lassez, and Maher (1995), who propose a concept for selecting suitable axes for separating polygons so as to improve the quality of the filter step. Basically, their work refines an earlier work by Jagadish (1990). However, although much of the work done in the area of spatial databases can be used readily for constraint database system, there are numerous research issues to be addressed. For example, spatial indices are mostly only suitable for indexing low dimensional data (Berchtold, Keim, and Kriegel 1996). Since constraint database systems the indices are likely to index higher dimensional data, these index structures have to be reconsidered as to whether they can be applied for indexing constraints. Furthermore, it is not clear how to index in the presence of disjunctions of conjunctions or how to use indices for nested constraints. The latter issue has been addressed by Brodsky and Wang (1995), who developed a first framework. However, it may be worth having a look at the methods proposed for indexing complex objects for object-oriented databases (Bertino 1994), to see how they can be integrated. Freeston (1995a) proposes to use BV-trees (Freeston 1995b) for indexing constraints. What makes BV-trees particularly interesting is their worst guarantee of the number of page accesses for exact match searches and that it possible to adapt this idea to other structures such as the BANG file. That multidimensional indexing methods quite nicely integrate into a constraint logic environment is shown by Bressan, Sexton, and Wallace (1996). They outline an implementation of an multidimensional indexing method within a constraint logic setting.

4.5 Query Optimization and Processing

Another aspect that has proven to be critical for the performance of relational database systems is query optimization. Declarative query languages are designed to be independent of any query evaluation plan. For one given query there are typically several different ways to evaluate it, each of which has a different performance. Since finding the optimal evaluation strategy is NP-hard, the art of query optimization is to find a close to optimal evaluation strategy in reasonable time. In relational database the issue of query optimization and processing has attracted a great deal of attention, resulting in a large body of literature (for surveys see Jarke and Koch (1984, Graefe (1993, Mishra and Eich (1992)). The issue of query optimization has also been addressed in the area of deductive (Bancilhon and Ramakrishnan 1986) and object-oriented databases (**?**; **?**).

It is not clear whether conventional optimization strategies such as algebraic transformation and simplification, cost-based plan selection, pushing down selections, join ordering, and magic sets can be adapted to serve in a constraint setting. For example, heuristics such as applying selections as early as possible during evaluation (also know as the push down heuristics) assume that the constraints are available at optimization time. This, however, is often not possible in constraint database systems. Moreover the cost of projection and selection are assumed to be small in comparison with the cost of retrieving a tuple or a data object in a traditional database system. In a constraint database this is frequently not be the case (Brodsky, Jaffar, and Maher 1993), since, for example, projection in a constraint database involves quantifier elimination. This paper studies optimization and processing of queries for linear constraint databases. In contrast to the model of Kanellakis et al. they allow existentially quantified constraints in constraint tuples. In order to obtain the values necessary for their cost-based two-phase optimization, they perform sampling.

Variations of magic set optimization (Ramakrishnan 1991) have also been studied for constraint database systems or deductive systems enhanced with constraints. These techniques essentially work as follows: The constraints derived during the first step, are propagated in the second step to ensure their early application (Mumick et al. 1990; Srivastava and Ramakrishnan 1992; Levy and Sagiv 1992; Kemp and Stuckey 1993; Stuckey and Sudarshan 1994).

Helm, Marriott, and Odersky (1995) study optimization of spatial queries composed of Boolean constraints. Their optimization decomposes a given query, composed of n spatial variables into a sequence of one variable range queries. Bressan (1994) and Wallace et al. (1995) study the use of well-known techniques from the area of constraint logic programming such as forward checking for optimizing queries.

Gaede and Günther (1995a) show how to use constraints for processing user-defined functions as they occur in object-oriented databases. By means of additional user-defined functions, which are defined with respect to the query predicate, they dynamically precompute constraints on query attributes. These constraints, in turn, can be used by the underlying database system to evaluate the query.

Not only traditional optimization mechanisms have to be newly regarded with respect to their applicability in constraint database systems, but also query processing strategies. In particular, traditional join processing techniques (Mishra and Eich 1992) have to be adapted to the special needs of constraint database systems (Brodsky, Jaffar, and Maher 1993) and new techniques have to be devised.

Although constraint databases pose some new problems for query optimization, because constraint relations are a true generalization of traditional relations, they contribute to a simpler view of the optimization problem. Such a simplification became possible for logic programming when it was extended to CLP.

The reason is that constraints provide a concise and clear representation of the state of the query evaluation. Retrieved constraints - extracted from generalized tuples - are added to the query constraints during query evaluation. Moreover constraints can also be added during progress down the index tree (see Section 4.4). As shown in Section 2.3, these extracted constraints are divided into two groups: basic constraints and constraint agents. The *basic constraints* are the simpler ones whose consistency can be checked efficiently. The *constraint agents* are the more complex constraints whose processing is generally more expensive. Consistency checking and quantifier elimination on constraint agents may be postponed until some more basic constraints have arrived which enable the constraint agents to simplify.

At any time during query evaluation the following can occur:

- Basic constraints are reduced to canonical form, and consistency is checked

- A constraint agent may use basic constraints to simplify itself and produce further basic constraints

An example class of basic constraints is the class of bounded real intervals; an example class of constraint agents is the class of non-linear polynomial constraints. Polynomial constraints can be used to propagate intervals, but it is probably not worth attempting to establish their global consistency.

The basic constraint solver is now both the mechanism for guiding search down the index tree, and for rejecting data items that do not satisfy the query. The constraint agents offer a form of dynamic optimization, and they can propagate information necessary for guiding the search in exactly the way that is needed to efficiently solve the introductory example of Brodsky, Jaffar, and Maher (1993).

4.6 Object-Oriented Features

Constraint database systems also have the potential to integrate object-oriented features such as complex objects, object-identifiers, class hierarchies, user-defined functions, and the like. However, there seems to be no fundamental problem in combining constraint databases with object-oriented concepts, since there orthogonal and several proposals have been made how to combine them.

Perhaps the first major work in this area was done at Bellcore, where logic rules, axioms and constraints were handled in an object-oriented database by a bottom-up demon mechanism (Caseau 1989). The demons were produced by translating the rules into an algebra, and compiling the algebraic terms into low-level demons. This work yielded the object-oriented, constraint language *Laure* (Caseau 1990).

Complex object have attracted some attention in the area of constraint database systems (Kanellakis and Goldin 1994; Srivastava et al. 1994; Grumbach and Su 1995; Revesz 1995). The key idea of complex objects is to have a number of type constructors (e.g., tuple or (finite) set constructor) which allow users to build up more complex objects such as sets of tuples composed of set-valued attributes (Abiteboul, Hull, and Vianu 1994).

The idea of having constraint object-oriented constraint query languages is put forward by Srivastava et al. (1994) and Brodsky and Kornatzky (1995). While the former proposal is more concerned with theoretical aspects of such a language, the latter focuses on how the integrate constraints into the XSQL framework (Kifer, Kim, and Sagiv 1992). As the work by Revesz (1995) shows, constraints are also useful in the presence of of class hierarchies. Here constraints lead to much cleaner and succinct way of describing certain aspects than currently possible in other systems.

4.7 Systems

The preceding discussion has reflected some of the theoretical research into constraint databases. Indeed there has been good progress, and many insightful theoretical results have emerged. By contrast the practical results we have presented, were largely in the constraint programming area. There is now a strong pressure to complement the theoretical research with some implementation work in the area of constraint database. This would help to get a better understanding of issues related to the implementation of these systems. Such systems could also function as a testbed for various promising application areas, to see whether constraint database systems are able to deliver their promises. Conversely, practical problems resulting from applications could also lead to further theoretical research into various aspects of constraint database systems.

Basically, there are three different approaches for building constraint database systems and each of them has its counterpart in the history of database systems:

1. *Database centered view.* Augment current database system (e.g., object-

oriented, deductive, and extended relational) by the notion of constraints. Most of these systems are specifically designed to be extensible (?; ?) and should therefore offer a good basis for experimentation. However, such an undertaking entails that the various database components have to be carefully scrutinized as to whether they could benefit from constraints. In a second step, the impact of such an augmentation on other components has to be studied. Furthermore, new components capable of efficiently dealing with constraint have to be introduced. This approach finds its counterpart in extended (nowadays called object-relational) database systems (?).

2. *Constraint programming centered view.* Augment current constraint programming system by the notion of persistence, transactions, indexing, etc. To achieve a tight coupling, the obvious way is to enhance a programming language (or system) by database statements. This approach has been used to build object-oriented databases, where some of the most popular object-oriented programming languages have been enhanced in such a way (for example Smalltalk in Gemstone). For constraint database systems, a first example is the Megalog database extension of ECLiPSe (?). The ECLiPSe language handles linear constraints, interval constraints and finite domain constraints, as well as concurrent constraints and a sublanguage for building specialised constraint handlers. Whilst Megalog (Bocca 1992) can store constrained variables, and supports multi-dimensional indexing, it lacks an integrated constraint query language, indexing and optimisation.

3. *Mixed view.* Since each of the above two approaches emphasizes a certain aspect (e.g., database or constraint programming system), there is an inherent risk that the complexity of the corresponding other part is going to be underestimated. Building a constraint database system from scratch could avoid this risk, but it certainly is the hardest. The DISCO (Datalog with Integer and Set order COnstraints) system (Byon and Revesz 1995) is a recent example of a system following along this line. In order to retain the set-at-a-time query evaluation for constraint database systems, DISCO uses the following strategy:

   (a) the given query is translated into a constraint algebra, whose operators work on graphs composed of a gap-graph (Revesz 1993) and set-graph (Revesz 1995)

   (b) after applying certain operators to the graph, the query is evaluated.

   The C^3 system (Brodsky 1996) on the other hand is built on existing components. This system couples the object-oriented database system ObjectStore (?) with the CPLEX linear constraint solver (CPLEX 1993). It is designed with extensibility in mind and aims at functioning as generic platform for building systems such as Lyric (Brodsky and

Kornatzky 1995) or a constraint extension of an object-oriented query language.

The C^3 system introduces the Constraint Comprehension Calculus, which builds constraints into *monoid comprehensions*. These comprehensions are designed to support clean database query optimisation techniques.

Finally some loose and some tight connections have been built between database systems and constraint programming platforms. For example the commercial CHIP - Constraint Handling in Prolog - system has an Oracle connection (**?**). ECLiPSe has also been connected to Oracle and to the Exodus storage manager (**?**; **?**).

5 Summary and Future Directions

In this paper, we gave an informal introduction to a new but promising area of database research: constraint database systems. These systems have been inspired by constraint programming and first formal definition were put forward back in 1990. Since then this area has flourished and recent research has resulted into a number of interesting theoretical contributions to the understanding of such systems. Important results delineating the expressive power of constraint query languages have been established, and establishing the data complexity of queries and constraint databases containing different classes of constraints.

The modelling potential of constraints in databases has been explored, suggesting that constraints offer modelling facilities orthogonal to, and compatible with, those offered by complex objects. Moreover this modelling power does not seem to be bought at the expense of efficient query evaluation. On the contrary, the integration of constraints invoked in the query, used as discriminators in the database directories, and stored in the database itself, would seem to support very powerful and clean optimisation facilities.

The applications that are emerging for constraint databases include geographic information systems, temporal databases, distributed databases, and hybrid databases storing data of different kinds, as required in environmental databases for example. Constraint databases can support the specialised requirements of these applications within a general database framework, thus obviating the need for specialised systems of the kind commercially available today.

Constraint databases nicely complement object databases. Their integration - dubbed "constraint object databases" - offer four major benefits over object databases.

1. They are built on a firm mathematical foundation. The lack of such a solid mathematical understanding is perceived as one of the greatest disadvantages of object-oriented systems.

2. Constraint query language are high level, declarative and highly expressive. This makes them not only suitable as query languages, but

also as “database programming languages”. Moreover the user of a constraint language can express even complex user-defined functions in a common framework, without having to resort to imperative programming. This has important benefits for the optimisation of queries.

3. Constraint database systems have the potential of truly removing the impedance mismatch, between programming and database access. Since constraints are not only used as universal data type but query and programming language, there is no need for copying one representation into the other. Object databases have the same benefit when accessing objects by identifier, but not when accessing data objects via their properties.

4. Constraints of a natural representaion of continuous data, and avoid redundancy in the storage of objects.

Whilst the benefits of constraint databases have been spelled out in detail, the technology has yet to tested out at the coalface! The proof of the pudding is in the eating, and now there is a pressing need to substantiate our claims with a substantial working implementation.

There are numerous practical issues which need to be addressed.

1. Can the greater expressive power, naturalness and flexibility of constraint databases be obtained without paying a price in performance? Previous experience with high-level data models would suggest that they cannot, and only a working implementation can now establish the truth. The practical issue is how to develop a query optimiser that operates effectively on complex queries over large numbers of stored constraints.

2. This is a closely related issue. Dynamic query optimisation lies at the heart of constraint programming. Can it migrate to constraint databases? Can a one-at-a-time style of query evaluation be made to run efficiently against a database? Or conversely can concurrent constraint handling techniques be made to work in a set-at-a-time style?

3. Can constraints be stored compactly? Can constraints be stored with traditional data? What compromises will be necessary to store constraints so that they can be retrieved with an efficiency comparable to traditional data? What compromises will be necessary to allow constraints to be stored with traditional data?

4. Can constraints be indexed effectively? Can updates on constraints be handled both at the storage and at the index levels?

5. Can existing data migrate to constraint databases? Can users migrate to constraint query languages? Can constraint databases be embedded in software environments that don’t offer any support for constraints?

6. Which classes of constraints are useful for which applications? Which classes of constraints can be supported efficiently? And is there a non-empty intersection?

Many of these questions are being addressed by researchers, but they will remain open until being settled, one way or the other, by working implementations. Nevertheless, as this review has hopefully shown, answers are emerging which would suggest the technology will work in practice, and we have cause to be confident that constraint databases will finally come off the drawing board.

One technology to watch, we suggest, is interval constraint propagation. This technology is currently making great strides in the constraint programming world (for example Van Hentenryck, McAllester, and Kapur (1995)) and it fits hand-in-glove with database indexing techniques. Finally it naturally supports the storage of data intervals, instead of data points, which offers a very clean mechanism for dealing with uncertainty.

References

(1994). Query processing for advanced database systems.

(1995). Proceedings of the ILOG user-group meeting. Available from `http://www.ilog.com`.

Abiteboul, S., R. Hull, and V. Vianu (1994). *Foundations of Databases.* Addison-Wesley.

Afrati, F., S. Cosmadakis, S. Grumbach, and G. Kuper (1994). Expressiveness of linear vs. polynomial constraints in database query languages. In A. Borning (Ed.), *PPCP'94: Second Workshop on Principles and Practice of Constraint Programming*, Seattle WA.

Aggoun, A. and N. Beldiceanu (1993). Extending CHIP in order to solve complex scheduling and placement problems. *Journal of Mathematical and Computer Modelling 17*(7), 57–73.

Aschenbrenner, T., A. Brodsky, and Y. Kornatsky (1995). Constraint database approach to spatio-temporal fusion and sensor management. In *Proc. ILPS95 Workshop on Constraints, Databases and Logic Programming*, Portland, Oregon.

Azarmi, N. and W. Abdul-Hameed (1995). Workforce scheduling with constraint logic programming. *BT Technology Journal 13*(1).

Bancilhon, F. and R. Ramakrishnan (1986). An amateur's introduction to recursive query processing strategies. In *Proc. ACM Int. Conf. on Management of Data*, Washington, D.C., pp. 16–52.

Belussi, A., E. Bertino, M. Bertolotto, and B. Catania (1995). Generalized relational algebra: Modeling spatial queries in constraint databases. In G. Kuper and M. Wallace (Eds.), *Proc. 1st Int. Database Workshop on Constraint Database Systems (CDB'95)*, Friedrichshafen, Germany, pp. 40–67. LNCS No. 1034.

Benedikt, M., Guozhu, L. Libkin, and L. Wong (1996). Relational expressive power of constraint query languages. In *Proc. 15th ACM Symposium on Principles of Database Systems (PODS)*, Montreal, Canada, pp. 5 – 16.

Benhamou, F. and L. Granvilliers (1996). Combining local consistency, symbolic rewriting and interval methods. In J. Calmet, J. Campbell, and J. Pfalzgraf (Eds.), *Artificial Intelligence and Symbolic Mathematical Computation, AISMC-3*, pp. 144–159.

Benjamin, M., T. Viana, K. Corbett, and A. Silva (1993). Satisfying multiple-rated constraints in a knowledge based decision aid. In *Proc. IEEE Conf. on Artificial Intelligence Applications*, Orlando.

Berchtold, S., D. Keim, and H.-P. Kriegel (1996). The X-tree: An index structure for high-dimensional data. In *Proc. 22th Int. Conf. on Very Large Data Bases.*

Bertino, E. (1994). A survey of indexing techniques for object-oriented databases. In *(Freytag, Maier, and Vossen 1994)*, pp. 383–418.

Bocca, J. B. (1992). *MEGALOG a programming platform for constructing information systems.* MacGraw-Hill.

Bormann, J., J. Lohse, M. Payer, and R. Schmid (1995). Circuit verification environment cve2. Technical Report Version 1.72, ZFE BT SE 12 Siemens AG. (User Manual and System Description).

Borning, A. and B. Freeman-Benson (1995). The OTI constraint solver: A constraint library for constructing graphical user interfaces. In *Proc Principles and Practice of Constraint programming CP'95.* Springer Verlag.

Bressan, S. (1994). Database query optimization and evaluation as constraint satisfaction problem solving. In *Workshop on Constraints and Databases*, Ithaca, NY, USA, pp. 27 – 40. Available as technical report from the University of Nebraska UNL- CSE-94-025.

Bressan, S., A. Sexton, and M. Wallace (1996). Implementing index structures using constraint logic programming. In *CP96 Workshop on Constraints and Databases*, Boston. Available as a technical report from George Mason University.

Brodsky, A. (1996). Constraint databases: Promising technique or just intellectual exercise. Available under URL http://www.cs.brown.edu/people/pvh/group.html.

Brodsky, A., D. Q. Goldin, and V. E.Segal (1996). On strongly polynomial projections in d-monotone constraint databases. In *CP96 Workshop on Constraints and Databases*, Boston. Available as a technical report from George Mason University.

Brodsky, A., J. Jaffar, and M. J. Maher (1993). Toward practical constraint databases. In *Proc. 19th Int. Conference on Very Large Data Bases (VLDB)*, Dublin, pp. 567 – 580.

Brodsky, A. and Y. Kornatzky (1995). The LyriC language: Querying constraint objects. In *Proc. ACM SIGMOD Conference on Management of Data*, San Jose, CA, pp. 35–46.

Brodsky, A., C. Lassez, J.-L. Lassez, and M. J. Maher (1995). Seperability of polyhedra for optimal filtering of spatial and constraint data. In *Proc. 14th ACM Symposium on Principles of Database Systems (PODS)*, San Jose, CA, pp. 54 – 64.

Brodsky, A., V. Segal, and P. Exarkhopoulou (1996). The c^3 constraint object-oriented database system. Technical Report, Dept. Information and Software Systems Engineering, George Mason University.

Brodsky, A. and X. S. Wang (1995). On approximation-based query evaluation, expensive predicates and constraint objects. In *Proc. Workshop on Constraints, Databases and Logic Programming*, Portland, OR.

Buchberger, B. (1985). Gröbner Bases: an Algorithmic Method in Polynomial Ideal Theory. In N. K. B. Ed. (Ed.), *Multidimensional Systems theory*, pp. 184–232. D. Reidel Publishing Company, Dordrecht - Boston - Lancaster.

Büettner, W. and H. Simonis (1987, October). Embedding boolean expressions into logic programming. *Journal of Symbolic Computation 4*, 191–205.

Burgard, W., A. B. Cremers, T. H. Kolbe, and L. Plümer (1996). Object construction by deduction for a 3D-geo-information system of a mine. In *Proc. Practical Applications of Prolog, PAP'96*, London.

Byon, J.-H. and P. Z. Revesz (1995). DISCO: A cnstraint database with sets. In G. Kuper and M. Wallace (Eds.), *Proc. 1st Int. Database Workshop on Constraint Database Systems (CDB'95)*, Friedrichshafen, Germany, pp. 68–83. LNCS No. 1034.

Carlier, J. and E. Pinson (1994). Adjustment of heads and tails for the job-shop problem. *European Journal of Operations Research 78*, 146–161.

Caseau, Y. (1989). A formal system for producing demons and rules. In *Proc. of the 1st Conf. on Deductive and Object-Oriented Databases DOOD'89*, Kyotot.

Caseau, Y. (1990, October). A deductive object-oriented language. *Annals of Mathematics and Artificial Intelligence*.

Caseau, Y. and F. Laburthe (1994). Improved clp scheduling with task intervals. In P. Van Hentenryck (Ed.), *Proceedings of the 11th International Conference on Logic Programming*, Santa Margherita. MIT Press.

Chandra, A. K. and D. Harel (1980). Computable queries for relational data bases. *Journal of Computer and System Science 21*(2), 156–178.

Chomicki, J., D. Q. Goldin, and G. M. Kuper (1996). Variable independence and aggregation closure. In *Proc. 15th ACM Symposium on*

Principles of Database Systems (PODS), Montreal, Canada, pp. 40 – 48.

Chomicki, J. and G. Kuper (1995). Measuring infinite relations. In *Proc. 14th ACM Symposium on Principles of Database Systems (PODS)*, San Jose, CA, pp. 78 – 85.

Clark, D. A., C. J. Rawlings, J. Shirazi, L.-L. Li, M. Reeve, K. Schuerman, and A. Veron (1993). Solving large combinatorial problems in molecular biology using the elipsys parallel constraint logic programming system. *The Computer Journal 36*(8), 690–701.

Codd, E. F. (1970). A relational model for large shared data banks. *Communications of the ACM 13*(6), 377–387.

Codd, E. F. (1972). Relational completeness of database sublanguages. In R. Rustin (Ed.), *Data Base Systems*, pp. 65–98. Prentice Hall.

Cohen, J. (1990). Constraint logic programming languages. *Communications of the ACM 33*(7), 52 – 68.

Collins, G. E. and H. Hong (1991). Partial Cylindrical Algebraic Decomposition for Quantifier Elimination. *Journal of Symbolic Computation 12*, 299–328.

Cooper, E. C. (1982). On the expressive power of query languages for relational databases. In *Proc. First ACM Symp. on Principles of Database Systems*, Los Angeles.

COSYTEC (1994, November). Proceedings of the CHIP users club. COSYTEC SA, Parc Club Orsay Universite, 4, rue Jean Rostand, 91893 Orsay Cedex, France.

CPLEX (1993). Using the cplex callable library and cplex mixed integer library. Technical Report Version 2.1, CPLEX Optimisation Inc.

DeGr (1986). Logic programming : Functions, relations and equations.

Donikian, S. and G. Hegron (1994). *Constraint Management in a Declarative Design Method for 3D Scene Sketch Modelling.* In Saraswat and Van Hentenryck Saraswat and Van Hentenryck (1994).

Doursenot, S., D. Clark, C. J. Rawlings, and A. Veron (1993). Contig mapping using elipsys. In *Proc. Workshop on "AI and the Genome", IJCAI*, Chambery, France.

Dowling, W. and J. H. Gallier (1984). Linear-time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming 1.*

Feldman, J., A. A. Hoyos, N. Sekas, and D. Vergamini (1995). Scheduling engine for LILCO. In *Proc. ILOG Solver Users Conference*, Paris. Also available on http://www.ilog.com.

Freeston, M. (1995a). The application of multi-dimensional indexing methods to constraints. In G. Kuper and M. Wallace (Eds.), *Proc. 1st*

Int. Database Workshop on Constraint Database Systems (CDB'95), Friedrichshafen, Germany, pp. 102–119. LNCS No. 1034.

Freeston, M. (1995b). A general solution of the n-dimensional B-tree problem. In *Proc. ACM SIGMOD Conference on Management of Data*, pp. 80–91.

Freuder, E. (1978, November). Synthesizing constraint expressions. *Communications of the ACM 21*, 958–966.

Fromherz, M. P. J. and V. A. Saraswat (1995). Model-based computing: Constructing constraint-based software for electro-mechanical systems. In *Proc. Practical Applications of Constraint Technology, PACT'95*.

Gaede, V. and O. Günther (1995a). Constraint-based query optimization and processing. In G. Kuper and M. Wallace (Eds.), *Proc. 1st Int. Database Workshop on Constraint Database Systems (CDB'95)*, Friedrichshafen, Germany, pp. 84–101. LNCS No. 1034.

Gaede, V. and O. Günther (1995b). Survey on multidimensional access methods. Technical Report ISS-15, Humboldt-Universität zu Berlin, Germany. submitted.

Gaede, V. and W.-F. Riekert (1996). Query evaluation in the object-oriented GIS GODOT. Technical Report ISS-30, Humboldt-Universität zu Berlin, Germany. submitted.

Gervet, C. (1994). Conjunto: Constraint logic programming with finite set domains. In *Proc. ILPS'94*, pp. 339–358.

Gleicher, M. (1994). *Practical Issues in Graphical Constraints.* In Saraswat and Van Hentenryck Saraswat and Van Hentenryck (1994).

Goldin, D. and P. C. Kanellakis (1996). Constraint query algebras. *Constraints 1*(1).

Goldin, D. Q. and P. Kanellakis (1995). On similarity queries for time-series data: Constraint specification and implementation. In U. Montanari and F. Rossi (Eds.), *Proc. 1st Int. Conference on Principles and Practice of Constraint Programming (CP'95)*, Cassis, France, pp. 137 – 153. Springer–Verlag. LNCS 976.

Goldstein, J., R. Ramakrishnan, and J.-B. Yu (1996). Using constraints to query R*-trees. In *Workshop on Constraints and Databases.*

Graefe, G. (1993). Query evaluation techniques for large databases. *ACM Computing Surveys 25*(2), 73–170.

Grumbach, S. and J. Su (1995). Dense-order constraint databases. In *Proc. 14th ACM Symposium on Principles of Database Systems (PODS)*, San Jose, CA, pp. 66 – 77.

Grumbach, S. and J. Su (1996). Towards practical constraint databases. In *Proc. 15th ACM Symposium on Principles of Database Systems (PODS)*, Montreal, Canada, pp. 28 – 37.

Günther, O. (1989). The cell tree: An object-oriented index structure for geometric databases. In *Proc. 5th Int. Conference on Data Engineering*, pp. 598–605.

Günther, O. and V. Gaede (1996). Oversize shelves: A storage management technique for large spatial data objects. *Int. Journal of Geographic Information Systems*.

Gupta, V., R. Jagadeesan, V. Saraswat, and D. Bobrow (1995). Programming in hybrid constraint languages. In P. Antsaklis, W. Kohn, A. Nerode, and S. Sastry (Eds.), *Hybrid Systems II*, Number 999 in Lecture Notes in Computer Science, pp. 552 – 567. Springer Verlag.

Heintze, N. and J. Jaffar (1994). Set constraints and set-based analysis. In A. Borning (Ed.), *Proc. 2nd Workshop on Principles and Practice of Constraint Programming*, Seattle WA. Invited talk presented by Joxan Jaffar.

Helm, R., T. Huynh, C. Lassez, and K. Marriott (1992). A linear constraint technology for interactive graphic systems. In *Proc. Graphics Interface '92*, Vancouver.

Helm, R., K. Marriott, and M. Odersky (1995). Spatial query optimization: From Boolean constraints to range queries. *Journal of Computer and System Science 51*, 197–210.

Hermosilla, L. and G. Kuper (1995). Towards the definition of a spatial object-oriented data model with constraints. In G. Kuper and M. Wallace (Eds.), *Proc. 1st Int. Database Workshop on Constraint Database Systems (CDB'95)*, Friedrichshafen, Germany, pp. 120–131. LNCS No. 1034.

Hermosilla, L. H. (1993). A requirement analysis for supporting temporal reasoning in geographic information systems. Technical Report ECRC-93-15, ECRC, Arabellastr 17, München.

Heus, K. and G. Weil (1996). Constraint programming: a nurse scheduling application. In *Proc. Practical Applications of Constraint Technology, PACT'96*, London, pp. 115–128.

Huynh, T. and C. Lassez (1990). An expert decision-support system for option-based investment. *Computer Mathematics with Applications 20*(9/10), 1–14.

Hyvőnen, E. (1991). Interval constraint spreadsheets for financial planning. In *Proc. First International Conference on Artificial Intelligence Applications on Wall Street.* IEEE Computer Society Press.

Jaffar, J. and J.-L. Lassez (1986, August). Constraint logic programming. IBM, Thomas J. Watson Research Center, Yorktown Heights, USA.

Jaffar, J. and J.-L. Lassez (1987, January). Constraint logic programming. In *Proceedings of the 14th ACM POPL Symposium*, Munich, West Germany.

Jaffar, J. and M. J. Maher (1994). Constraint logic programming: A survey. *Journal of Logic Programming 19,20*, 503–581.

Jagadish, H. V. (1990). Spatial search with polyhedra. In *Proc. 6th Int. Conference on Data Engineering*, pp. 311–319.

Jagadish, H. V. (1991). A retrieval technique for similar shapes. In *Proc. ACM SIGMOD Conference on Management of Data*, pp. 208 – 217.

Jagadish, H. V. and A. O. Mendelzon (1995). Similarity-based queries. In *Proc. 14th ACM Symposium on Principles of Database Systems (PODS)*, San Jose, CA.

Jain, R. (1993). Report on the NSF workshop on visual information management systems.

Jarke, M. and J. Koch (1984). Query optimization in database systems. *ACM Computing Surveys 16*, 111–152.

Kanellakis, P., S. Ramaswamy, D. Vengroff, and J. Vitter (1993). Indexing for data models with constraints and classes. In *Proc. 12th ACM Symposium on Principles of Database Systems (PODS)*, Washington, DC, pp. 233–243.

Kanellakis, P. C. and D. Q. Goldin (1994). Constraint programming and database query languages. Technical report, Brown University, Providence.

Kanellakis, P. C., G. Kuper, and P. Revesz (1990). Constraint query languages. In *Proc. 9th ACM Symposium on Principles of Database Systems (PODS)*, Nashville, Tennessee, pp. 299–313.

Kanellakis, P. C., G. Kuper, and P. Revesz (1995). Constraint query languages. *Journal of Computer and System Science 51*(1), 26–52.

Kay, P. and H. Simonis (1995). Building industrial CHIP applications from reusable software components. In *Proc. Conf. on Practical Applications of Prolog, PAP'95*.

Kemp, D. and P. Stuckey (1993). Analysis-based constraint query optimization. In D. S. Warren (Ed.), *Proc. 10th Int. Conference on Logic Programming*, pp. 666–682. MIT Press.

Kifer, M., W. Kim, and Y. Sagiv (1992). Querying object-oriented databases. In *Proc. ACM SIGMOD Conference on Management of Data*, pp. 393–402.

Koubarakis, M. (1993). Representing and querying in temporal databases: The power of temporal constraints. In *Proc. 9th Int. Conference on Data Engineering*.

Koubarakis, M. (1994). Foundations of indefinite constraint databases. In A. Borning (Ed.), *PPCP'94: Second Workshop on Principles and Practice of Constraint Programming*, Seattle WA.

Koubarakis, M. (1996). Tractable disjunctions of linear constraints. In *Principles and Practice of Constraint Programming, CP'96*, Boston. LNCS 1118.

Kowalski, R. A., F. Sadri, and P. Soper (1987). Integrity checking in deductive databases. In *Proc. Thirteenth Int. Conf. on Very Large Data Bases*, Brighton, U.K., pp. 61–69.

Kuper, G. M. (1993). Aggregation in constraint databases. In *Proc. 1st Workshop on Principles and Practice of Constraint Programming (PPCP'93):*.

Lazaro, J. M. and P. Aristondo (1996). Job rostering with constraints. In *Proc. Practical Applications of Constraint Technology, PACT'96*, London, pp. 155–168.

Levy, A. and Y. Sagiv (1992). Constraints and redudancy in Datalog. In *Proc. 12th ACM Symposium on Principles of Database Systems (PODS)*, San Diego, CA, pp. 67 – 80.

Mailharro, D. and T. Le Quenven (1995). A constraint-based tool for automatic sizing of an instrument and control architecture. In *Proc. ILOG Solver Users Conference*, Paris. Also available on http://www.ilog.com.

Mezzanatto, G., M. Foglino, P. Giordanengo, M. Apra, and G. Gullane (1993). Using ai techniques to design and install electrical bundles. In *Proc. 13th International Conf. on Artificial Intelligence, Expert Systems and Natural Language*, Avignon. EC2, 269-287, rue de la Garenne, 92024 Nanterre, France.

Miranker, D. P., R. J. Bayardo, and V. Samolados (1996). Query evaluation as constraint search: An overview of early results. In *CP96 Workshop on Constraints and Databases*, Boston. Available as a technical report from George Mason University.

Mishra, P. and M. H. Eich (1992). Join processing in relational databases. *ACM Computing Surveys 24*(1), 63–113.

Mohr, R. and T. C. Henderson (1986). Arc and path consistency revisited. *Artificial Intelligence 28*, 225–233.

Monfroy, E. (1992, August). Gröbner Bases: Strategies and Applications. In *Conference on Artificial Intelligence and Symbolic Mathematical Computations*, Karlsruhe, Germany.

Montanari, U. (1974). Networks of constraints: Fundamental properties and applications to picture processing. *Information Science 7*(2), 95–132.

Müller, J. (1995). Environmental monitoring system. ESPRIT Project 6757.

Mumick, I. S., S. J. Finkelstein, H. Pirahesh, and R. Ramakrishnan (1990). Magic conditions. In *Proc. 9th ACM Symposium on Principles of Database Systems (PODS)*, Nashville, Tennessee, pp. 314 – 330.

Pavlopoulou, C., A. P. Gionis, P. Stamatopoulos, and C. Halatsis (1996). Crew pairing optimisation based on CLP. In *Proc. Practical Applications of Constraint Technology, PACT'96*, London, pp. 191–210.

Puddicombe, A. (1996). National land information service.

Puget, J.-F. (1995). Applications of constraint programming. In U. Montanari and F. Rossi (Eds.), *Proc. 1st Int. Conference on Principles and Practice of Constraint Programming (CP'95)*, Cassis, France. Springer–Verlag. LNCS 976.

Puget, J.-F. (1996). Finite set intervals. Presented at the Workshop on Set Constraints, CP'96.

Ramakrishnan, R. (1991). Magic templates. *Journal of Logic Programming 11*, 189–216.

Revesz, P. Z. (1993). A closed form evaluation for datalog queries with integer (gap-) order constraints. *Theoretical Computer Science 116*(1), 117 – 149.

Revesz, P. Z. (1995). Datalog queries of set constraint databases. In *Proc. Int. Conference on Database Theory*, pp. 425 – 438.

Revesz, P. Z. (1996). Genomic databse applications in DISCO. In *CP96 Workshop on Constraints and Databases*, Boston. Available as a technical report from George Mason University.

Roman, G.-C. (1990). Formal specification of data processing requirements. *IEEE Transactions on Knowledge and Data Engineering 2*(12), 370–380.

Saraswat, V. (1993). *Concurrent Constraint Programming*. Logic Programming and Doctoral Dissertion Award Series. MIT Press.

Saraswat, V. J. and P. Van Hentenryck (Eds.) (1994). *Principles and Practice of Constraint Programming*. MIT Press.

Schiwietz, M. (1993). *Speicherung und Anfragebearbeitung komplexer Geo-Objekte*. Ph. D. thesis, Ludwig-Maximilians Universität München. In German.

Shankar, A., D. Gilbert, and M. Jampel (1996). Transient analysis of linear circuits using constraint logic programming. In *Proc. Practical Applications of Constraint Technology, PACT'96*, London, pp. 221–248.

Simonis, H. (1995). Application development with the CHIP system. In G. Kuper and M. Wallace (Eds.), *Proc. 1st Int. Database Workshop on Constraint Database Systems (CDB'95)*, Friedrichshafen, Germany, pp. 1–21. LNCS No. 1034.

Simonis, H. and T. Cornelissens (1995). Modelling producer/consumer constraints. In U. Montanari and F. Rossi (Eds.), *Proc. 1st Int. Conference on Principles and Practice of Constraint Programming (CP'95)*, Cassis, France. Springer–Verlag. LNCS 976.

Srivastava, D. and R. Ramakrishnan (1992). Pushing constraint selection. In *Proc. 11th ACM Symposium on Principles of Database Systems (PODS)*, San Diego, CA, pp. 301– 315.

Srivastava, D., R. Ramakrishnan, and P. Revesz (1994). Constraint objects. In A. Borning (Ed.), *PPCP'94: Second Workshop on Principles and Practice of Constraint Programming*, Seattle WA.

Stolboushkin, A. P. and M. A. Taislin (1996). Linear vs. order constraint queries over rational databases. In *Proc. 15th ACM Symposium on Principles of Database Systems (PODS)*, Montreal, Canada, pp. 17 – 27.

Stonebraker, M. (Ed.) (1994). *Readings in Database Systems*, San Mateo. Morgan Kaufmann. Second edition.

Stuckey, P. J. and S. Sudarshan (1994). Comiling query constraints. In *Proc. 13th ACM Symposium on Principles of Database Systems (PODS)*, Minneapolis, MA, pp. 56 – 67.

Tarski, A. (1951). *A Decision Method for Elementary Algebra and Geometry*. Berkeley: University of California Press. 2nd revised edition.

The Practical Application Company (1996). *Practical Application of Constraint Technology*. The Practical Application Company. `proceedings@pap.com`.

Topor, R. W. (1987). Domain independent formulas and databases. *Theoretical Computer Science 52*(3), 281–306.

van Emden, M. H. and R. A. Kowalski (1976). The semantics of predicate logic as a programming language. *Journal of the ACM 23*(4), 733–742.

Van Gelder, A. and R. W. Topor (1987). Safety and correct translation of relational calculus formulas. In *Proc. Sixth ACM Symp. on Principles of Database Systems*, San Diego, Ca., pp. 313–327. Revised version to appear in *ACM Transactions on Database Systems*.

Van Hentenryck, P. (1989a). A Logic Language for Combinatorial Optimization. *Annals of Operations Research: Special Issue on Links with Artificial Intelligence*.

Van Hentenryck, P. (1989b). *Constraint Satisfaction in Logic Programming*. Logic Programming Series. Cambridge, MA: MIT Press.

Van Hentenryck, P., D. McAllester, and D. Kapur (1995). Solving polynomial systems using a branch and prune approach. *SIAM Journal on Numerical Analysis*.

Wallace, M. G. (1991, August). Compiling integrity checking into update procedures. In *Proc. IJCAI'91*, Sydney.

Wallace, M. G. (1995). Constraints as a tool for distributed scheduling. In *Proc. International Conf on Improving Manufacturing Performance in the Distributed Enterprise: Advanced Systems and Tools*, pp. 129–146.

Wallace, M. G. (1996). Practical applications of constraint programming. *Constraints Journal 1*(1).

Wallace, M. G., S. Bressan, and T. L. Provost (1995). Magic checking: Constraint checking for database query optimisation. In G. Kuper and

M. Wallace (Eds.), *Proc. 1st Int. Database Workshop on Constraint Database Systems (CDB'95)*, Friedrichshafen, Germany, pp. 148–166. LNCS No. 1034.

Yakhno, T. M., G. B. Cheblakov, and V. Z. Zilberfain (1995). FISSURE: Finder of solutions with subdefinite resources. In *Proc. Contessa Workshop on Constraint Databases and Applications*, Friedrichshafen. Springer, LNCS Number 1034.

Zhang, Y. and A. K. Mackworth (1995). Synthesis of hybrid constraint-based controllers. In P. Antsaklis, W. Kohn, A. Nerode, and S. Sastry (Eds.), *Hybrid Systems II*, Number 999 in Lecture Notes in Computer Science, pp. 552 – 567. Springer Verlag.