# CSP Techniques for Solving Combinatorial Queries within Relational Databases

Malek Mouhoub and Chang Feng

University of Regina
Wascana Parkway, Regina, SK, Canada, S4S 0A2
{mouhoubm,feng202c}@cs.uregina.ca

**Abstract.** A combinatorial query is a request for tuples from multiple relations that satisfy a conjunction of constraints on tuple attribute values. Managing combinatorial queries using the traditional database systems is very challenging due to the combinatorial nature of the problem. Indeed, for queries involving a large number of constraints, relations and tuples, the response time to satisfy these queries becomes an issue. To overcome this difficulty in practice we propose a new model integrating the Constraint Satisfaction Problem (CSP) framework into the database systems. Indeed, CSPs are very popular for solving combinatorial problems and have demonstrated their ability to tackle, in an efficient manner, real life large scale applications under constraints. In order to compare the performance in response time of our CSP-based model with the traditional way for handling combinatorial queries and implemented by MS SQL Server, we have conducted several experiments on large size databases. The results are very promising and show the superiority of our method comparing to the traditional one.

## 1 Introduction

It is generally acknowledged that relational databases have become the most significant and the main platform [1] to implement data management and query searching in practice. The efficiency and veracity for query solving is acquiring more concerns and has been improved among various database systems. However, along with the complex requirement for query from the customer, a problem is triggered: the conventional relational database system often fails to obtain answers to the combinatorial query efficiently. Indeed, current Relational Database Management Systems (RDBMS) require a lot of time effort in answering complex combinatorial queries. [2] has given an accurate definition of a combinatorial query: *a request for tuples from multiple relations that satisfy a conjunction of constraints on tuple attribute values*. The following example illustrates the above definition.

**Example 1.** Let us assume we have a relational database containing three tables describing computer equipments: CPU (*CPU_id*, Price, Frequency, Quality), Memory (*Memory_id*, Price, Frequency, Quality), and Motherboard (*Motherboard_id*, Price, Frequency, Quality).

N.T. Nguyen and E. Szczerbicki (Eds.): Intel. Sys. for Know. Management, SCI 252, pp. 131–151.
springerlink.com 

| *CPU_id* | Price | Freq | Quality |
|---|---|---|---|
| 1 | 200 | 5 | 80 |
| 2 | 500 | 10 | 80 |
| 3 | 1100 | 8 | 90 |
| 4 | 2000 | 15 | 95 |

| *Mem_id* | Price | Freq | Quality |
|---|---|---|---|
| 1 | 1000 | 7 | 80 |
| 2 | 900 | 9 | 90 |
| 3 | 800 | 10 | 90 |
| 4 | 1000 | 15 | 100 |

| *Mboard_id* | Price | Freq | Quality |
|---|---|---|---|
| 1 | 100 | 6 | 50 |
| 2 | 200 | 6 | 50 |
| 3 | 300 | 11 | 80 |
| 4 | 400 | 15 | 70 |

```
Select CPU_id, Memory_id,Motherboard_id
From CPU, Memory, Motherboard  Where
 CPU.Price + Memory.Price + Motherboard.Price < 1500
 And CPU.Frequency + Memory.Frequency + Motherboard.Frequency < 30
 And CPU.Quality + Memory.Quality + Motherboard.Quality < 250
```

**Fig. 1.** An example of a combinatorial query.

Figure 1 illustrates the three tables and an example of a related combinatorial query. Here the combinatorial query involves a conjunction of three arithmetic constraints on attributes of the three tables.

Although a variety of query evaluation technologies such as hash-join [3], sorted-join, pair-wise join [4,1] have been implemented into the relational model attempting to optimize searching process, the effect is extremely unsatisfactory since most of them concentrate on handling simple constraints, not the complex ones [2,3,5,6]. For instance, in example 1 the arithmetic query involves a conjunction of three constraints that should be satisfied together. This requires higher complexity and selectivity [6] since the traditional model has to perform further complex combinatorial computations after searching for each sub query (corresponding to each of these 3 constraints) which will cost more time effort. In order to overcome this difficulty in practice, we propose in this paper a model that integrates the Constraint Satisfaction Problem (CSP) framework in the relational database system. Indeed, CSPs are very powerful for representing and solving discrete combinatorial problems. In the past three decades, CSPs have demonstrated their ability to efficiently tackle large size real-life applications, such as scheduling and planning problems, natural language processing, business applications and scene analysis. More precisely, a CSP consists of a finite set of variables with finite domains, and a finite set of constraints restricting the possible combinations of variable values [7,8]. A solution tuple to a CSP is a set of assigned values to variables that satisfy all the constraints. Since a CSP is known to be an NP-hard problem in general[1], a backtrack search algorithm of exponential time cost is needed to find a complete solution. In order to overcome this difficulty in practice, constraint propagation techniques have been proposed [7,9,11,10]. The goal of theses techniques is to reduce the size of the search space before and during the backtrack search. Note that some work on CSPs and databases has already been proposed in the literature. For instance, in [12] an analysis of the similarities between database theories and CSPs has been conducted. Later, [2] has proposed a general framework for modeling the combinatorial aspect, when dealing

[1] There are special cases where CSPs are solved in polynomial time, for instance, the case where a CSP network is a tree [9,10].

with relational databases, into a CSP. Query searching has then been improved by CSP techniques. Other work on using CSP search techniques to improve query searching has been reported in [13,14,15].

In this paper, our aim is to improve combinatorial query searching using the CSP framework. Experimental tests comparing our model to the MS SQL Sever demonstrate the superiority in time efficiency of our model when dealing with complex queries and large databases.

In the next section we will show how can the CSP framework be integrated within the RDBMS. Section 3 describes the details of our solving method. Section 4 provides the structure of the RDBMS with the CSP module. The experimental tests we conducted to evaluate the performance of our model are reported in section 5. We finally conclude in section 6 with some remarks and possible future work.

## 2 Mapping a Database with the Combinatorial Query into a CSP

There are many similarities between the structure of a database and the CSP model. Each table within the database can be considered as a relation (or a CSP constraint) where the table attributes correspond to the variables involved by the constraint. The domain of each variable corresponds to the range of values of the corresponding attribute. The combinatorial query is here an arithmetic constraint of the related CSP. For instance, the database of example 1 with the combinatorial query can be mapped into the CSP defined as follows.

- **Variables:**
  - CPU.Price,
  - CPU.Frequency,
  - CPU.Quality,
  - Memory_id.Price,
  - Memory_id.Frequency,
  - Memory_id.Quality,
  - Motherboard_id.Price,
  - Motherboard_id.Frequency
  - and Motherboard_id.Quality.
- **Domains:** Domains of the above variables.
  - CPU.Price: $\{200, 500, 1100, 2000\}$.
  - CPU.Frequency: $\{7, 9, 10, 15\}$.
  - CPU.Quality: $\{80, 90, 95\}$.
  - Memory_id.Price: $\{800, 900, 1000\}$.
  - Memory_id.Frequency: $\{7, 9, 10, 15\}$.
  - Memory_id.Quality: $\{80, 90, 100\}$.
  - Motherboard_id.Price: $\{100, 200, 300, 400\}$.
  - Motherboard_id.Frequency: $\{6, 11, 15\}$.
  - Motherboard_id.Quality: $\{100, 200, 300, 400\}$.

- **Constraints:**
  - **Semantic Constraints:**
    - CPU(*CPU_id*, Price, Frequency, Quality)
    - Memory(*Memory_id*, Price, Frequency, Quality)
    - Motherboard(*Motherboard_id*, Price,Quality)
  - **Arithmetic constraint:**

```
CPU.Price + Memory.Price + Mother.Price < 1500
And CPU.Frequency + Memory.Frequency + Motherboard.Frequency < 30
And CPU.Quality + Memory.Quality + Motherboard.Quality < 250
```

After this conversion into a CSP, we use CSP search techniques, that we will describe in the next section, in order to handle the combinatorial query in an efficient manner.

## 3 Solving Method

As we mentioned earlier, solving a CSP requires a backtrack search algorithm of exponential time cost. In order to overcome this difficulty in practice, local consistency techniques have been proposed [7,9,11,10]. The goal of theses techniques is to reduce the size of the search space before and during the backtrack search. More precisely, local consistency consists of checking to see if a consistency is satisfied on a subset of variables. If the local consistency is not successful then the entire CSP is not consistent. In the other case many values that are locally inconsistent will be removed which will reduce the size of the search space. Various local consistency techniques have been proposed [7]: node consistency which checks the consistency according to unary constraint of each variable, arc consistency which checks the consistency between any subset sharing one constraint, path consistency which checks the consistency between any subset sharing two constraints ..., etc.

In this paper we use arc consistency in our solving method. More formally, arc consistency [7] works as follows. Given a constraint $C(X_1,\ldots,X_p)$ then arc consistency is enforced through $C$ on the set $X_1,\ldots,X_p$ as follows. Each value $v \in D(X_i)$ $(1 \leq i \leq n)$ is removed if it does not have a support (value such that $C$ is satisfied) in at least one domain $D(X_j)$ $(1 \leq j \leq n$ and $j \neq i)$. $D(X_i)$ and $D(X_j)$ are the domains of $X_i$ and $X_j$ respectively.

Our method is described as follows.

1. First, arc consistency is applied on all the variables to reduce their domains. If a given variable domain is empty then the CSP is inconsistent (this will save us the next phase) which means that the query cannot be satisfied.
2. A backtrack search algorithm using arc consistency [7,9] is then performed as follows. We select at each time a variable and assign a value to it. Arc consistency is then performed on the domains of the non assigned variables in order to remove some inconsistent values and reduce the size of the search space. If one variable domain becomes empty then assign another value to the current variable or backtrack to the previously assigned variable in order to assign another value to this

latter. This backtrack search process will continue until all the variables are assigned values in which case we obtain a solution (satisfaction of the combinatorial query) or the entire search space has been explored without success. Note that during the backtrack search variables are selected following the *most constrained first* policy. This rule consists of picking the variable that appears in the largest number of constraints.

Before we present the details of the arc consistency algorithm, let us see with the following examples how arc consistency, through arithmetic and semantic constraints, is used to reduce the domains of the different attributes in both phases of our solving method.

**Example 2.** Let us consider the database of example 1. The domain of the variable CPU.Price is equal to $\{200, 500, 1100, 2000\}$. Using arc consistency with the subquery `CPU.Price + Memory.Price + Mother.Price < 1500`, we can remove the values 1100 and 2000 from the domain of CPU.price. Indeed there are no values in the domains of Memory.Price and Mother.Price such that the above arithmetic constraint is satisfied for 1100 and 2000.

**Example 3.** Let us now take another example where each attribute has a large number of values within a given range. For example let us consider the following ranges for the attributes of the table CPU.

```
200 <= Price <= 2000.
5 <= Frequency <= 15.
80 <= Quality <=95.
```

Using the above range for Price and the subquery:

```
CPU.Price + Memory.Price + Mother.Price < 1500
```

we can first reduce, the range of the attribute CPU.Price as follows:

```
200 <= CPU.Price < 1500.
```

When using the attributes Memory.Price and Motherboard.Price we reduce the upper bound of the above range as follows.

```
CPU.Price < 1500 - [Min(Memory.Price) + Min(Motherboard.Price)]
          < 1500 - [800 + 100]
          < 600
```

We will finally obtain the following range for CPU.Price:

```
200 <= CPU.Price < 600.
```

During the second phase of our solving method arc consistency is used through arithmetic and semantic constraints. The following example shows how an assignment of a value to a variable, during backtrack search, is propagated using arc consistency through the arithmetic constraint to update the domains (ranges) of the non assigned variables which will reduce the size of the search space.

**Example 4.** Let us consider the arithmetic constraint $A+B+C<100$, where the domains of A, B and C are respectively: [10, 80], [20, 90] and [20, 70]. Using arc consistency we will reduce the domains of A and B as follows. $10<A<100-(20+20)=60$. $20<B<100-(10+20)=70$. Suppose now that, during the search, we first assign the value 50 to $A$. Through arc consistency and the above arithmetic constraint, the domain of $B$ will be updated as follows: $20<B<100-(50+20)=30$.

**Example 5.** Let us consider the previous example and assume that we have the following semantic constraints (corresponding to three tables):
Table1($A$, $D$, ...),
Table2($B$, $E$, ...)
and Table3($C$, $F$, ...).

After assigning the value 50 to $A$ as shown above in example 4, the domain of $B$ will be reduced to [20,30]. Now through the semantic constraint Table1 all the tuples where $A \neq 50$ should be removed from Table1. Also, through Table2 all tuples where $B \notin$ [20,30] will be removed from Table2. This will reduce the domains of $D, E$ and $F$.

Arc consistency is enforced with an arc consistency algorithm. In the past three decades several arc consistency algorithms have been developed. However most of these algorithms deal with binary constraints [11,16,17,18,19,20]. In the case of n-ary constraints there are three types of algorithms [7].

- Generalized Arc Consistency: for general non binary constraints.
- Global constraints: for constraints involving all the variables of a given CSP (such as the constraint $allDifferent(X_1,\ldots,X_n)$ which means that the values assigned to each of the variables $X_1,\ldots,X_n$ should be mutually different).
- Bounds-Consistency: which is a weaker (but less expensive) form of arc consistency. It consists of applying arc consistency to the upper and lower bounds of the variable domains. Examples 3 and 4 above use a form of bounds-consistency which motivates our choice of this type of algorithm for arithmetic constraints as shown below.

In our work we use the improved generalized arc consistency (GAC) algorithm [21] for semantic constraints (since this algorithm is dedicated for positive table constraints) and the bound consistency algorithm for discrete CSPs [22] in the case of arithmetic constraints. More precisely, bounds consistency is first used through arithmetic constraints to reduce the bounds of the different domains of variables. The improved GAC [21] is then used through the semantic constraints to reduce the domains of the attributes

**Algorithm GAC**

1. Given a constraint network $CN = (X, C)$
   (*X: set of variables, C: set of constraints between variables*)
2. $Q \leftarrow \{(i,j) \mid i \in C \wedge j \in vars(C)\}$
3. ($vars(C)$ is the list of variables involved by $C$)
4. **While** $Q \neq Nil$ **Do**
5. $Q \leftarrow Q - \{(i,j)\}$
6. **If** $REVISE(i,j)$ **Then**
7. **If** $Domain(j) = Nil$ **Then** return false
8. $Q \leftarrow Q \sqcup \{(k,l) \mid k \in C \wedge j \in vars(k)$
9. $\wedge\ l \in vars(k) \wedge k \neq i \wedge j \neq l\}$
10. **End-If**
11. **End-While**
12. Return true

**Function** $REVISE(i,j)$
(**REVISE for bound consistency**)

1. $domainSize \leftarrow |Domain(j)|$
2. **While** $|Domain(j)| > 0$
   $\wedge \neg seekSupportArc(i,j,min(j))$ **Do**
3. remove $min(j)$ from $Domain(j)$
4. **End-While**
5. **While** $|Domain(j)| > 1$
   $\wedge \neg seekSupportArc(i,j,max(j))$ **Do**
6. remove $max(j)$ from $Domain(j)$
7. **End-While**
8. Return $domainSize \neq |Domain(j)|$

**Function** $REVISE(i,j)$
(**REVISE for handling semantic constraints**)

1. $REVISE \leftarrow false$
2. $nbElts \leftarrow |Domain(j)|$
3. **For** each value $a \in Domain(j)$ **Do**
4. **If** $\neg seekSupport(i,j,a)$ **Then**
5. remove $a$ from $Domain(j)$
6. $REVISE \leftarrow true$
7. **End-If**
8. **End-For**
9. Return Revise

**Fig. 2.** GAC algorithm and Revise for bound consistency (bottom left) and for handling semantic constraints (bottom right).

even more. Let us describe now the details of our method. The basic GAC algorithm [23,21] is described in figure 2. This algorithm enforces the arc consistency on all variables domains. GAC starts with all possible pairs $(i,j)$ where $j$ is a variable involved by the constraint $i$. Each pair is then processed, through the function *REVISE* as follows. Each value $v$ of the domain of $j$ should have a value supporting it (such that the constraint $j$ is satisfied) on the domain on every variable involved by $i$ otherwise $v$ will be removed. If there is a change in the domain of $j$ (after removing values without support) after calling the function *REVISE* then this change should be propagated to all the other variables sharing a constraint with $j$.

When used with arithmetic constraints (as a bound consistency algorithm) $C$ contains the list of arithmetic constraints and the *REVISE* function (the function that does the actual revision of the domains) is defined as shown in figure 2 [22]. In the other case where GAC is used with semantic constraints $C$ contains these arithmetic constraints and the *REVISE* function is defined as shown in the bottom right of figure 2 [21]. In the function *REVISE* (for bound consistency) of figure 2, the function *seekSupportArc* (respectively thefunction *seekSupport* of *REVISE* for semantic constraints in figure 2) is called to

find a support for a given variable with a particular value. For instance when called in line 2 of the function *REVISE* for bound consistency, the function *seekSupportArc* looks, starting from the lower bound of $j$'s domain, for the first value that has a support in $i$'s domain. When doing so, any value not supported will be removed.

## 4 Structure of the RDBMS with the CSP Module

Figure 3 presents the architecture of the module handling combinatorial queries. In the CSP module, the CSP converter translates the query and the other information obtained

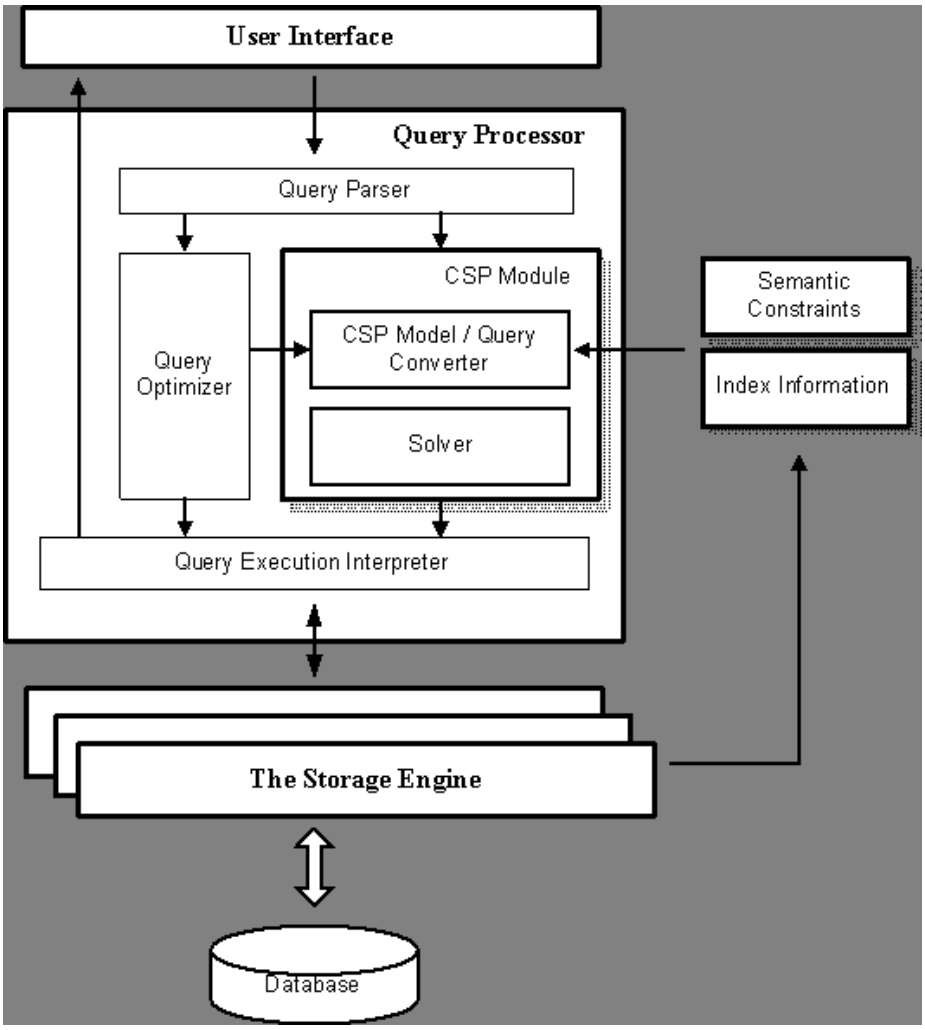



**Fig. 3.** Relational database model with the CSP module.

for the database such as the semantic constraints into a CSP. The solver contains the solving techniques we described in the previous section.

More precisely, in the traditional RDBMS model the query submitted by the user through the user interface is first received by the query parser which will transform the string form of the query into query objects that will be handled by the query processor. Regularly, the query optimizer will choose the appropriate query solving plan (query execution order with the least estimated cost) and send the execution request to the interpreter. The interpreter will then handle the request and implements the input/output operations by driving storage engine.

We address the function and working process of every component indicated in Figure 3 as follows.

**User Interface.** This is the interface being responsible to communicate with "front stage" such as website or other applications which have needs for data operations or retrievals. For instance, RDBMS like Microsoft SQL server provides a common user interface by setting connection string and corresponding parameters for .Net, PHP or Java platform.

**Query Processor.** The query processor is the most significant part in RDBMS that will handle query construction and optimization once the query is approached from the User Interface. It has to engender query execution plan sending to the next part of the system. The traditional model includes query parser, query optimizer and query execution interpreter. For our improved RDBMS model, the CSP module will be added into the system to deal with combinational query solving.

**Query Parser.** The query parser is the component in the query processor which will create the query to other solvers of RDBMS according to the requirement of query from User.

**Query Optimizer.** The query optimizer is one of the key parts in the query processor that attempts to determine the more efficient query execution plan of the given query. Usually, the query optimizer is called as cost-based because it mainly considers system resource like memory and CPU cost as well as I/O operations to estimate the cost for the query execution and then select the best one to implement. The join algorithm [63] (pair-wise join, hash join or sorted merge join) will be formed in the query optimizer supported with indexing.

**Query Execution Interpreter.** This part takes responsibility of translating the query plan into the language which can be understood by the Storage Engine. Meanwhile, the solution returned from the database will be posted back to the application by the query execution interpreter.

**CSP Module.** The new module added into RDBMS to solve combinational query has two sub components: CSP Query Converter and CSP Solver. The CSP Query Converter will convert the combinational query into the CSP which can be accessed and solved by the CSP Solver. Since some combinational query still need structural adjustment or

join order optimization, for some cases the query should be optimized by the query optimizer before it is sent to the CSP query parser. Moreover, some database technologies and information will support the CSP query parser to reduce the size of the search problem such as indexing and semantic constraints.

**Storage Engine.** The Storage Engine implements related operations with data in database based on the order submitted by the query processor.

In the case of the RDBMS model integrating the CSP module, once the combinatorial query is parsed by the query parser, it is sent to the CSP module. The converter will convert the query (including the arithmetic constraints) and the semantic constraints obtained from the database into the CSP. The solver will then solve it and send the solution to the execution interpreter.

### 4.1 CSP Module Design

The class diagram in Figure 4 gives the details of how the CSP module is designed to handle combinational queries. The sequence diagram in Figure 5 indicates the working steps of the CSP module during the search.

The query parser gains the query by using the GetQuery function from the application. The system will decide to activate the query optimizer or not. Because sometimes the order of query can be optimized that will increase the efficiency of search, but sometimes it cannot. After index and semantic constraints information are obtained by the Query Converter, the optimized combinational query will be converted into a CSP. Continuously, the QueryAnalyze function is run to analyze the CSP and decide which specific consistency checking strategy will be used to prune the domains of the CSP. The solving method we presented in the previous section will then be applied to find a solution to the query. The Solver will then call the query interpreter to send request to the lower component to carry out the compilation.

The sequence diagram in Figure 5 shows how CSP module solves the combinational query step by step based on the main functions made in the Figure 4. Sometimes the

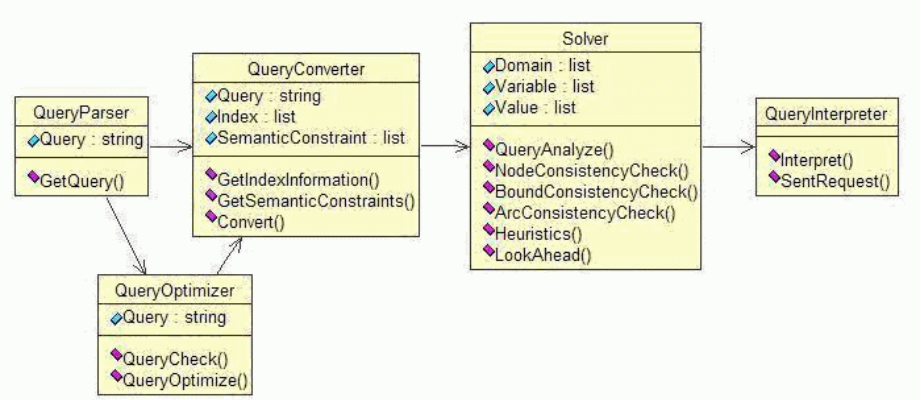



**Fig. 4.** Class Diagram for the CSP module.

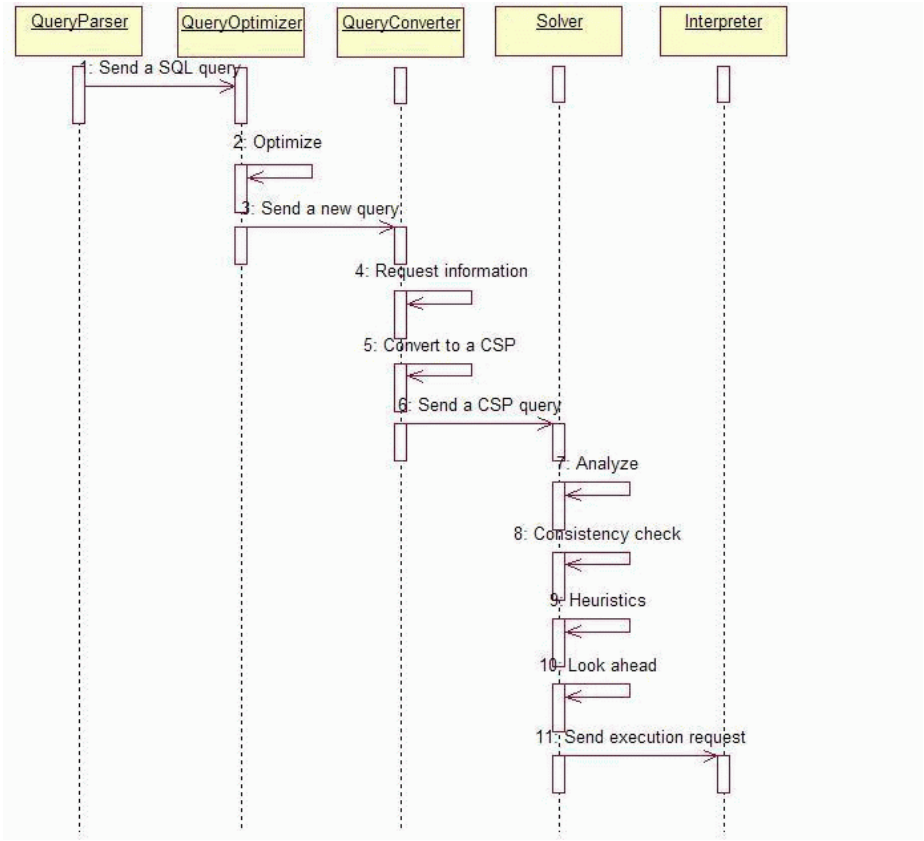



**Fig. 5.** Sequence Diagram for the CSP module.

query optimizer will not be activated during the search process, since it is not a necessary part for solving combinational query. The relations among it and other components have been described in Figures 4 and 5. In the following we will provide two concrete examples to show the whole solving process.

## 4.2 Examples

### 4.2.1 Computer Components Selection

We take a combinational query as an example to introduce the solving process in the CSP module as shown in Figure 4. Suppose the data source for the combinational query in Figure 6 is the one shown in Figure 1.

After optimization by the query optimizer, the second sub query is removed from the combinational query by using the merge sorted join algorithm [26]. The reason is although the constraint for the first and second sub query is all about the sum of computer components price, the arithmetic constraint condition of the first one is smaller than the second, which means the first sub query requires the smaller variable domain

```
Select CPU_id, Memory_id, Motherboard_id
  From CPU, Memory, Motherboard Where
    CPU.Price + Memory.Price + Motherboard.Price <1500 And
    CPU.Price + Memory.Price + Motherboard.Price <1600 And
    CPU.Frequency + Memory.Frequency + Motherboard.Frequency <32
    And CPU.Quality + Memory.Quality + Motherboard.Quality < 250
```

**Fig. 6.** A SQL combinational query.

```
Select CPU_id, Memory_id, Motherboard_id
  From CPU, Memory, Motherboard Where
    CPU.Price + Memory.Price + Motherboard.Price <1500 And
    CPU.Frequency  + Memory.Frequency + Motherboard.Frequency <32 And
    CPU.Quality + Memory.Quality + Motherboard.Quality < 250
```

**Fig. 7.** The optimized SQL combinational query

for solving than the second. As a result, the condition for the combinational query is optimized as shown in Figure 4.

In the next step, the query converter will implement converting from a combinational query into a CSP. The optimized SQL combinational query in Figure 5 is converted as follows.

- **Variables:**
  - CPU_id,
  - CPU.Price,
  - CPU.Frequency,
  - CPU.Quality,
  - Memory_id,
  - Memory.Price,
  - Memory.Frequency,
  - Memory.Quality,
  - Motherboard_id,
  - Motherboard.Price,
  - Motherboard.Frequency,
  - Motherboard.Quality
- **Domains:** Domains of the above variables.
  - CPU_id [1, 4],
  - CPU.Price [200, 2000],
  - CPU.Frequency [5, 15],
  - CPU.Quality [80, 95],
  - Memory_id [1, 4],
  - Memory.Price [800, 1000],
  - Memory.Frequency [7, 15],
  - Memeory.Quality [80, 100],
  - Motherboard_id [1, 4],

- Motherboard.Price [100, 400],
- Motherboard.Frequence [6, 15],
- Motherboard.Quality [50, 80]

– **Constraints:**
  - **Semantic Constraints:**
    * CPU(*CPU_id*, Price, Frequency, Quality)
    * Memory(*Memory_id*, Price, Frequency, Quality)
    * Motherboard(*Motherboard_id*, Price,Frequency,Quality)
  - **Arithmetic constraint:**

```
CPU.Price + Memory.Price + Motherboard.Price < 1500
And CPU.Frequency + Memory.Frequency + Motherboard.Frequency < 32
And CPU.Quality + Memory.Quality + Motherboard.Quality < 250
```

Furthermore, index information (filtering information) needs to be collected for consistency checking of Solver. For example, the filtering information in Figure 1 is:

– CPU.Price [200, 2000],
– CPU.Frequency [5, 15],
– CPU.Quality [80, 95],
– Memory.Price [800, 1000],
– Memory.Frequency [7, 15],
– Memeory.Quality [80, 100],
– Motherboard.Price [100, 400],
– Motherboard.Frequence [6, 15],
– Motherboard.Quality [50, 80],

The above information could be applied for the node or bound consistency checking in the solving process. The semantic constraints shown above will be used for arc consistency checking.

Once all necessary information has been gained by the system, our solver will look for a possible solution. Finally, the interpreter will translate the solution found by the solver into the execution order, which can be understood by the storage engine and will be carried out by the compilation. One solution to the above example is:

– CPU (1, 200, 5, 80),
– Memory (1, 1000, 7, 80),
– Motherboard (1, 100, 6, 50).

#### 4.2.2 Vehicle Elements Selection

Let us take another example to explain the solving process. Since the solving process is similar to the previous example, only the result of each step will be represented in this example. The data source for the combinational query are shown in Figure 8.

To look for the available combinations of vehicle elements, the user creates a combinational query as follows.

| Engine_id | Price | Usage | Rate |
|---|---|---|---|
| 1 | 4000 | 11 | 8 |
| 2 | 5000 | 10 | 8.5 |
| 3 | 2000 | 8 | 9 |
| 4 | 5000 | 15 | 9 |

| Clutch_id | Price | Usage | Rate |
|---|---|---|---|
| 1 | 2000 | 5 | 7 |
| 2 | 1000 | 7 | 6 |
| 3 | 1100 | 8 | 5.5 |
| 4 | 1500 | 10 | 5 |

| Gear_id | Price | Usage | Rate |
|---|---|---|---|
| 1 | 1000 | 3 | 7 |
| 2 | 500 | 5 | 8 |
| 3 | 800 | 8 | 7 |
| 4 | 1000 | 10 | 6 |

**Fig. 8.** Tables of vehicle elements

```
Select Engine_id, Clutch_id, Gear_id
From Engine, Clutch, Gear Where
Engine.Price + Clutch.Price + Gear.Price >5500 And
Engine.Usage + Clutch.Usage + Gear.Usage <20 And
Engine.Rate + Clutch.Rate + Gear.Rate < 28
```

Unlike the combinational query in the previous example, there is no sub query that can be optimized or removed in the Query Optimizer. As a result, the combinational query is directly converted into a CSP by the query converter as follows.

- **Variables**
  - Engine_id,
  - Engine.Price,
  - Engine.Usage,
  - Engine.Rate,
  - Clutch_id,
  - Clutch.Price,
  - Clutch.Usage,
  - Clutch.Rate,
  - Gear_id,
  - Gear.Price,
  - Gear.Usage,
  - Gear.Rate
- **Domains**
  - Engine_id [1, 4],
  - Engine.Price [2000, 5000],
  - Engine.Usage [8, 15],

- Engine.Rate [8, 9],
- Clutch_id [1, 4],
- Clutch.Price [1000, 2000],
- Clutch.Usage [5, 10],
- Clutch. Rate [5, 7],
- Gear_id [1, 4],
- Gear.Price [500, 1000],
- Gear.Usage [3, 10],
- Gear.Rate [6, 8]

– **Constraints**
  - **Semantic Constraints:**
    * Engine(*Engine_id*, Price, Usage, Rate)
    * Memory(*Clutch_id*, Price, Usage, Rate)
    * Motherboard(*Gear_id*, Price,Usage,Rate)
  - **Arithmetic constraint:**

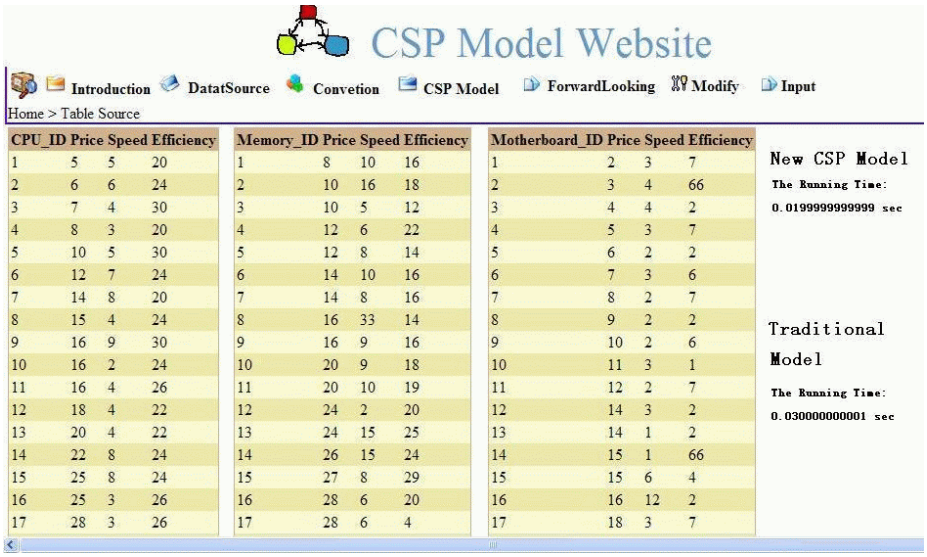
```
Engine.Price + Clutch.Price + Gear.Price < 5500
And Engine.Usage + Clutch.Usage + Gear.Usage < 20
And Engine.Rate + Clutch.Rate + Gear.Rate < 28
```

The index information is collected by RDBMS: Engine.Price [2000, 5000], Engine.Usage [8, 15], Engine.Rate [8, 9], Clutch.Price [1000, 2000], Clutch.Usage [5, 10], Clutch. Rate [5, 7], Gear.Price [500, 1000], Gear.Usage [3, 10], Gear.Rate [6, 8]. This will be applied in node and bound consistency checking to prune search space in the solving process. In the final step, the system will run a backtrack search to find the solution using the algorithm we presented earlier. One solution to this problem is Engine (3, 2000, 8, 9), Clutch (1, 2000, 8, 9), Gear (1, 1000, 3, 7).

## 5 Experimentation

In order to compare the time performance of our query processor with one of the most advanced relational databases (MS SQL server 2005) we run several tests on randomly generated databases and take the running time in seconds needed (by our method and MS SQL) to satisfy the query. The tests are conducted on a IBM T42 with a P4 1.7 GHz processor and 512 MB RAM memory, running Windows XP.

We have built a web-based application to simulate the CSP model and run the experiments. This application has been developed using ASP.NET (*C#*) connected with SQL server (See Figure 10). In the tests, we can compare the efficiency of operations from RDBMS and the simulated CSP model. At first, three tables are set up in the RDBMS, and the data in the tables are randomly created and input by data access layer of the application for tests. We create all the data creation, manipulation and solving classes in this layer, which is responsible to connect with the relational database and implement all data operations. CSP search algorithms, constraint propagation and heuristic methods are written in this layer, all of which achieve the functions of the CSP solver as shown in Figure 3. The size and number of tables, as well as the complexity of combinational queries are changed during the testing for the performance comparison between the old and new models. Index is added for the first non-primary key column in the table, in order to offer the acceleration for traditional search and ”filtering” information for consistency checking of the new CSP module.




| CPU_ID | Price | Speed | Efficiency |
|---|---|---|---|
| 1 | 5 | 5 | 20 |
| 2 | 6 | 6 | 24 |
| 3 | 7 | 4 | 30 |
| 4 | 8 | 3 | 20 |
| 5 | 10 | 5 | 30 |
| 6 | 12 | 7 | 24 |
| 7 | 14 | 8 | 20 |
| 8 | 15 | 4 | 24 |
| 9 | 16 | 9 | 30 |
| 10 | 16 | 2 | 24 |
| 11 | 16 | 4 | 26 |
| 12 | 18 | 4 | 22 |
| 13 | 20 | 4 | 22 |
| 14 | 22 | 8 | 24 |
| 15 | 25 | 8 | 24 |
| 16 | 25 | 3 | 26 |
| 17 | 28 | 3 | 26 |

| Memory_ID | Price | Speed | Efficiency |
|---|---|---|---|
| 1 | 8 | 10 | 16 |
| 2 | 10 | 16 | 18 |
| 3 | 10 | 5 | 12 |
| 4 | 12 | 6 | 22 |
| 5 | 12 | 8 | 14 |
| 6 | 14 | 10 | 16 |
| 7 | 14 | 8 | 16 |
| 8 | 16 | 33 | 14 |
| 9 | 16 | 9 | 16 |
| 10 | 20 | 9 | 18 |
| 11 | 20 | 10 | 19 |
| 12 | 24 | 2 | 20 |
| 13 | 24 | 15 | 25 |
| 14 | 26 | 15 | 24 |
| 15 | 27 | 8 | 29 |
| 16 | 28 | 6 | 20 |
| 17 | 28 | 6 | 4 |

| Motherboard_ID | Price | Speed | Efficiency |
|---|---|---|---|
| 1 | 2 | 3 | 7 |
| 2 | 3 | 4 | 66 |
| 3 | 4 | 4 | 2 |
| 4 | 5 | 3 | 7 |
| 5 | 6 | 2 | 2 |
| 6 | 7 | 3 | 6 |
| 7 | 8 | 2 | 7 |
| 8 | 9 | 2 | 2 |
| 9 | 10 | 2 | 6 |
| 10 | 11 | 3 | 1 |
| 11 | 12 | 2 | 7 |
| 12 | 14 | 3 | 2 |
| 13 | 14 | 1 | 2 |
| 14 | 15 | 1 | 66 |
| 15 | 15 | 6 | 4 |
| 16 | 16 | 12 | 2 |
| 17 | 18 | 3 | 7 |

**Fig. 9.** The user interface of new RDBMS with CSP module.

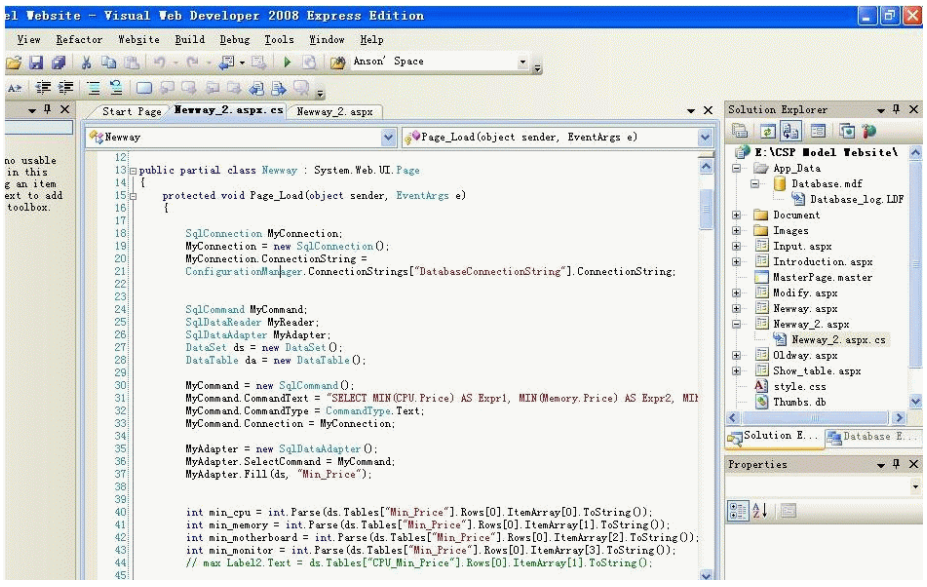



**Fig. 10.** The programming environment for the CSP module.

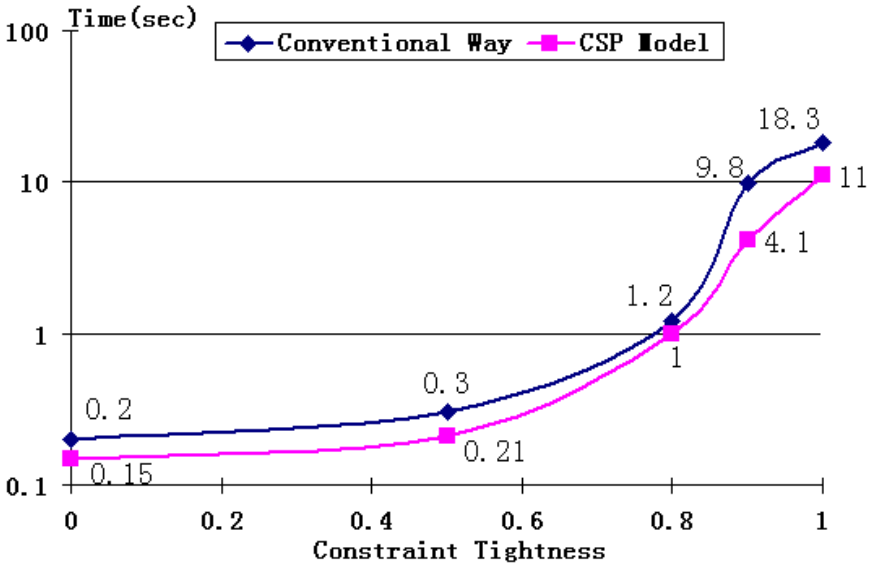



**Fig. 11.** Test results when varying the tightness.

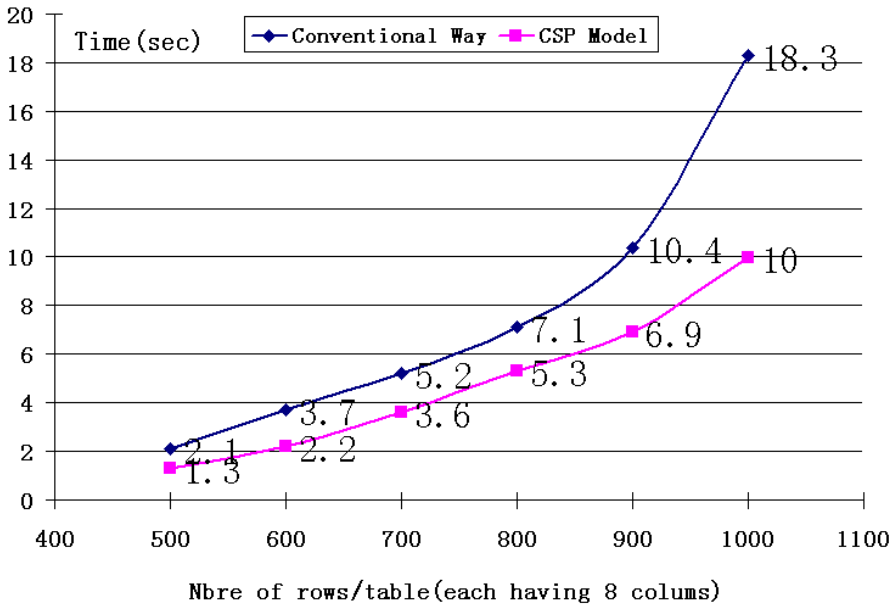



**Fig. 12.** Test results when varying the number of rows.

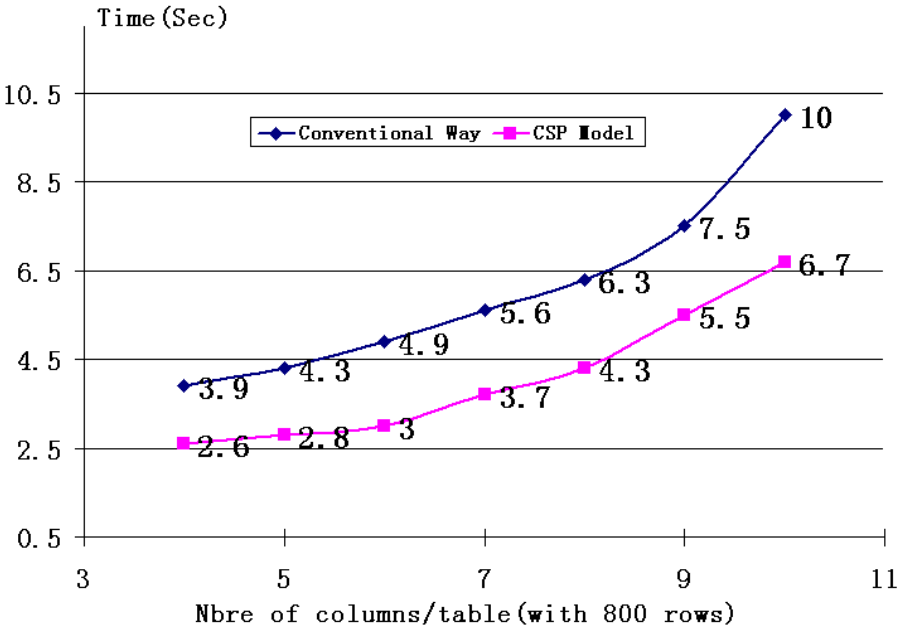



**Fig. 13.** Test results when varying the number of columns.

Figure 9 is the screen shot of the comparative result of the test on the combinational query of the example in Section 4.1. The three tables are displayed with the corresponding results from the traditional RDBMS model and our model as shown in the right. Figure 10 is the screen shot of the programming environment for the experiments.

Each database and its corresponding combinatorial query are randomly generated according to the following parameters: $T$ (the number of tables within the database), $R$ and $C$ (respectively the number of rows and columns of each table within the database); and $P$ the constraint tightness. This last parameter defines how easy|hard is the CSP corresponding to the generated database. More precisely, the constraint tightness $P$ of a given constraint is defined as "*the fraction of all possible pairs of intervals from the domains of the variables (involved by the constraint) that are not allowed by the constraint [24].*" According to this definition, the value of $P$ is between 0 and 1. Easy problems are those where the tightness is small and hard problems correspond to a high tightness value.

Figure 11 presents tests performed when varying the tightness $P$. $T$,$R$ and $C$ are respectively equal to 3, 800 and 8. Note that the logarithmic scale is used here for the y coordinates As we notice on the chart, when $T$ is below 0.8 (the case where %80 of the possibilities are not solutions) the 2 methods have similar running time. However when $T$ is more than 0.8 (which corresponds to hard problems where only few possibilities are solutions) we can see the superiority of our method over the traditional model. The other figures 14, 12, and 13 correspond to the situation where we vary $T$, $R$ and $C$ respectively. In all these 3 cases we can easily see that our method outperforms MS SQL (especially when $R$ or $T$ increases). Note that in the case where $T$ is greater than 3

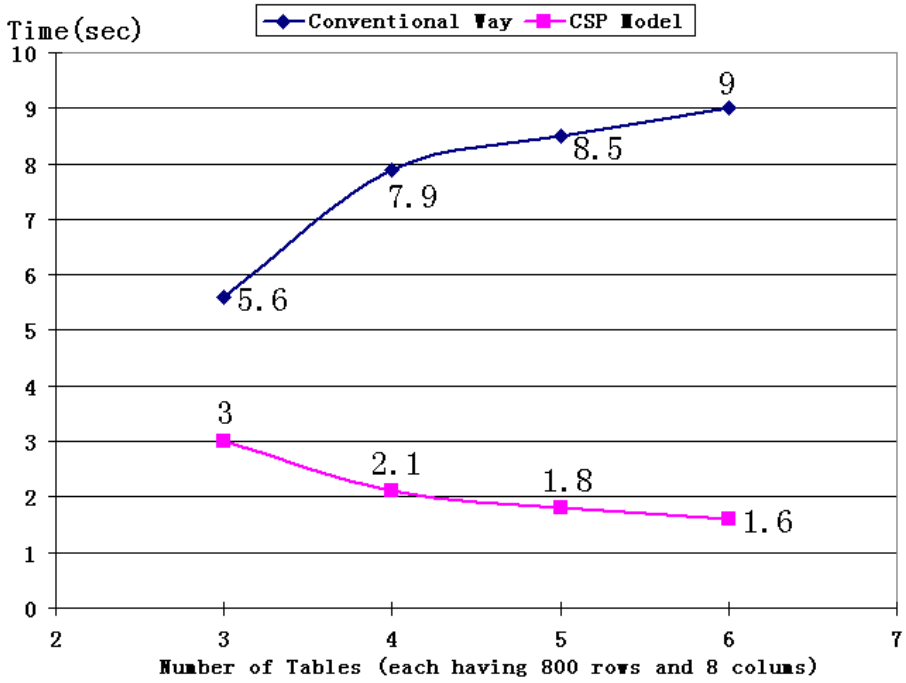



**Fig. 14.** Test results when varying the number of tables.

(in figure 14), the problem becomes inconsistent (the query cannot be satisfied). In this particular case, our method detects the inconsistency before the backtrack search (see step 1 of our solving method in Section 3). Since the backtrack search phase is saved in this case, the total running time is even better than the case where $T$ is equal to 3.

## 6 Conclusion and Future Work

CSPs are a very powerful framework to deal with discrete combinatorial problems. In this paper we apply CSPs in order to solve a particular case of combinatorial applications consisting of satisfying a combinatorial query. More precisely our method translates the combinatorial query with the database information into a CSP. CSP techniques including constraint propagation and backtrack search are then used to satisfy the query by looking for a possible solution to the CSP. In order to demonstrate the efficiency in practice of our method, we conducted different tests on large databases and compared the running time needed by our method and the well known SQL server 2000 in order to satisfy combinatorial queries. The results of the tests demonstrate the superiority of our method for all the cases.

In the near future, we intend to integrate other search methods such as local search techniques and genetic algorithms in order to speed up the query search process. Indeed, we believe that while these approximation algorithms do not guarantee a complete solution they can be very useful in case we want an answer within a short time.

Another problem we intent to explore is handling several combinatorial queries at the same time. The obvious way is to process them one by one using the method we have proposed in this paper. However, this might not be the best idea. May be it is better to first pre-process the set of queries using arc consistency and then (if the queries are arc consistent) we can proceed with solving these queries together. An experimental comparative study of both ways needs to be carried out in order to find out which one is better in terms of processing time.

## References

1. Atzeni, P., Antonellis, V.D.: Relational database theory. Benjamin-Cummings Publishing Co. (1993)
2. Liu, C., Foster, I.T.: A framework and algorithms for applying constraint solving within relational databases. In: W(C)LP 2005, pp. 147–158 (2005)
3. Revesz, P.: Introduction to Constraint Databases. Springer, New York (2002)
4. Chuang, L., Yang, L., Foster, I.T.: Efficient relational joins with arithmetic constraints on multiple attributes. In: IDEAS, pp. 210–220 (2005)
5. Vardi, M.Y.: The complexity of relational query languages. In: Annual ACM Symposium on Theory of Computing (1982)
6. Chandra, A.: Structure and complexity of relational queries. In: The 21st IEEE Symposium, pp. 333–347 (1980)
7. Dechter, R.: Constraint Processing. Morgan Kaufmann, San Francisco (2003)
8. Hentenryck, P.V.: Constraint Satisfaction in Logic Programming. MIT Press, Cambridge (1989)
9. Haralick, R., Elliott, G.: Increasing tree search efficiency for Constraint Satisfaction Problems. Artificial Intelligence 14, 263–313 (1980)
10. Mackworth, A.K., Freuder, E.: The complexity of some polynomial network-consistency algorithms for constraint satisfaction problems. Artificial Intelligence 25, 65–74 (1985)
11. Mackworth, A.K.: Consistency in networks of relations. Artificial Intelligence 8, 99–118 (1977)
12. Vardi, M.Y.: Constraint sastisfaction and database theory: A tutorial. In: PODS 2000 (2000)
13. Swami, A.: Optimization of large join queries: combining heuristics and combinatorial techniques. In: The 1989 ACM SIGMOD international conference on Management of data, pp. 367–376 (1989)
14. Jarke, M., Koch, J.: Query optimization in database systems. ACM Computing Surveys (CSUR), 111–152 (1984)
15. Miguel, I., Shen, Q.: Solution techniques for constraint satisfaction problems: Foundations. Artificial Intelligence Review 15(4) (2001)
16. Mohr, R., Henderson, T.: Arc and path consistency revisited. Artificial Intelligence 28, 225–233 (1986)
17. Bessière, C.: Arc-consistency and arc-consistency again. Artificial Intelligence 65, 179–190 (1994)
18. Bessière, C., Freuder, E., Regin, J.: Using inference to reduce arc consistency computation. In: IJCAI 1995, Montréal, Canada, pp. 592–598 (1995)
19. Zhang, Y., Yap, R.H.C.: Making ac-3 an optimal algorithm. In: Seventeenth International Joint Conference on Artificial Intelligence (IJCAI 2001), Seattle, WA, pp. 316–321 (2001)
20. Bessière, C., Régin, J.C.: Refining the basic constraint propagation algorithm. In: Seventeenth International Joint Conference on Artificial Intelligence (IJCAI 2001), Seattle, WA, pp. 309–315 (2001)

21. Lecoutre, C., Radoslaw, S.: Generalized arc consistency for positive table constraints. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 284–298. Springer, Heidelberg (2006)
22. Lecoutre, C., Vion, J.: Bound consistencies for the csp. In: Proceeding of the second international workshop Constraint Propagation And Implementation (CPAI 2005) held with the 10th International Conference on Principles and Practice of Constraint Programming (CP 2005), Sitges, Spain (September 2005)
23. Mackworth, A.K.: On reading sketch maps. In: IJCAI 1977, pp. 598–606 (1977)
24. Sabin, D., Freuder, E.C.: Contradicting conventional wisdom in constraint satisfaction. In: Proc. 11th ECAI, Amsterdam, Holland, pp. 125–129 (1994)
25. Ribeiro, C.C., Ribeiro, C.D., Lanzelotte, R.S.: Query optimization in distributed relational databases. Journal of Heuristics 65, 5–23 (1997)
26. Sagiv, Y.C., Arbor, A.: Optimization of queries in relational databases, pp. 1–10. UMI Research Press (1981)