In this paper, we introduce the concept "team" and incorporate the team query into data language SQL. Existing relational query languages use tuple variables of tables to specify required data. Queries that are expressed by these languages are used to get all the data which satisfy the specified conditions. While a team variable represents a subset of a table, a team query is to retrieve all the teams that meet certain constraints.

We introduce team qualifications and discuss their implementations. The search space of a team query may be very large. Therefore, instead of using conventional breadth-first search method, we adopt a depth-first search method for query processing. Thus, users can always get results, if one exists.

In decision support environments the selection of teams with extreme values like maximum or minimum is frequently needed. In some cases, finding such teams is NP-complete or NP-hard. Genetic algorithms are proposed and computational results are presented. We incorporate genetic algorithms in the query optimization algorithm. The approach makes it possible for a DBMS to respond to team queries in a predicate restricted amount of time.

1 INTRODUCTION

Relational database systems are being used in data processing community. Relational database systems (DBMSs) provide nonprocedural query languages such as SQL7, 8, 20 and set-at-a-time operators. Thus, the application programmers or users need only specify what he wants without regarding how the results are retrieved. This greatly reduce application development time.

However, relational query languages can not be used to express teams that meet certain team constraints. Let us take a simple team example, a baseball team coach is looking for a team of nine players from the database of a baseball team to play a baseball game. The constraints of this team are (1) there are nine players in each team, (2) there should be exactly one player for each defense position. The result of this query may have many teams. This query is different from the queries like "get all the players who can play the pitcher position." Each player in the result of the query meets the constraint "the player who can play the pitcher position." An application programmer or user needs to write procedural programs to solve such team problems.

Obviously, a team query is different from a query of relational database query languages such as SQL. SQL finds a set of individuals and all individuals in the set satisfy the condition which is specified in the query. An element of a set meets all the constraints, while an element of a team may only meet some constraint(s) of the team constraints.

A SQL query and a team query have two different aspects. The first difference is the number of results. The result of a SQL query is a set of tuples, if one exists. However, the result of a team query is a set of teams. The second difference is in the team constraints. A SQL query does not specify the set of team constraints while a team query must specify the set of team constraints. Conditions that are used in SQL are not enough to express team criteria.

Teams are common and well understood in modern society. For example, in a company, a manager needs to organize teams to perform certain jobs; in an exercise, a coach has to assign a team of players to play games; a foreign needs to assign a team to negotiate with other countries; and so on. Teams exist everywhere in the real-world. However, so far we did not find any relational database languages that address the team query in the research literature. We extend SQL to be able to express and process teams that meet certain constraints. Hence, users with knowledge of SQL can easily take advantage of the team retrieval capabilities without learning a new language.

Team queries have two categories: (1) to select an optimal team, (2) to select all possible teams. The selection of a project team with minimal cost is of the first category. Queries of this category wish to find a team which has a maximum or minimum value of a given expression such as the sum of the salaries in the whole prospective teams. The selection of all possible teams of nine players from a baseball team is of the second category.

In some cases, finding teams with extreme values like minimum or maximum is NP-complete or NP-hard. It may take a very long time to find an optimum solution. Genetic algorithms discussed in this paper are designed to find near-optimal solutions. We found they performed well, both on the computational effort and the quality of the solutions via a variety of test problems.

The processing of team queries is quite different from that of the conventional database queries. Conventional database query processing uses a tuple-by-tuple evaluation strategy, while team query processing uses a set-by-set evaluation strategy. The search space of a team query is all subsets of the set of tuples of a relation. The search space of a team query may be large. Therefore, instead of using conventional breadth-first search method, we adopt a depth-first search method for query processing. Thus, users can always get results, if one exists.

1.1 Related Research

Sets and set manipulation occur naturally in many applications. Database query languages that attempt to incorporate sets and set operators have been defined in recent years by Abiteboul and Grambach20, Kuper17, Berri et al.3, 4, Chen6, and Gavish and Segel10.

COL2 is a logic-based language for complex objects. In addition to tuple and set constructs, the language also provides so-called "data functions". LDL3, 4 is an attempt
to combine the benefits of logic programming with those of relational query languages. In LDL3, 4), sets and set operators are introduced into this language. In HiLog6), sets can be represented naturally by parameterized predicates. The work10) addresses the problem of optimizing queries that involve set operations (set queries) in a distributed relational database system. A set query is defined to be any query that can be represented as a sequence of relational operations followed by set operations between one set of tuples and a group of sets of tuples.

Although, the above languages support sets and set manipulation, however, they did not identify the concepts of team.

Genetic algorithms11, 12) are general purpose optimization algorithms (somewhat akin to simulated annealing in that sense). They were developed by Holland12) to search irregular, poorly characterized spaces. The genetic algorithm does not necessarily find an optimal solution to any one problem, but does find good solutions to problems that are resistant to most other known techniques. Holland was inspired by the example of population genetics and used crossover rather than mutation as the primary genetic operator. Thus, genetic search proceeds over a number of generations, with each generation represented by a population of current chromosomes. "Survival of the fittest" provides the pressure for populations to develop increasingly fit individuals. Genetic algorithms have been successfully applied to various combinatorial optimization problems. Algorithms have been proposed for optimizing complex queries in the past5, 14, 15, 16).

1.2 Overview of This Paper
This paper is organized as follows. In Section 2 the language is briefly described and a number of motivating examples are given. Section 3 defined basic and special team qualifications. In Section 4, we present the query processing techniques of basic and special team qualifications. Transformation rules for team qualifications and operations are stated in Section 5. Analysis of query processing techniques is presented in Section 6. The last section is the conclusion.

2 TEAM-ORIENTED QUERIES
In this section we first briefly describe the syntax of team-oriented queries and then present a number of motivating examples. For a detailed description of the language, one is referred to the work13).

2.1 Query Specification in Team-Oriented Query Language
Let \( U = \{ R_1, R_2, \ldots, R_p \} \) be the universal set of relations, and \( W = \{ A_1, A_2, \ldots, A_n \} \) be the universal set of attributes. A team is a subset of the set of tuples over the specified attributes of a relation that satisfies a set of qualifications and a set of team qualifications. More formally, a team can be defined as a 4-tuple:

\[
\text{TEAM} = (R, A', \Theta, s\Theta),
\]

where \( R \in U, A' \subseteq W \), \( \Theta \) is a set of qualifications, and \( s\Theta \) is a set of team qualifications. The set of tuples over the set of attributes \( A' \) of relation \( R \) constructs the domain of a team.

A qualification \( \Theta \) is made up of a number of clauses of the form:

\[
\text{Attr rel-op const value}
\]

where \( \text{Attr} \in W, \text{rel-op} \) is normally one of the operators \( (=, \geq, \leq, \neq, <, >) \), and const-value is a constant value. Clauses can be arbitrarily connected by the Boolean operators AND, OR, and NOT to form a general qualification. A qualification here corresponds to a selection condition of the SELECT operation.

A team qualification \( s\Theta \) is any qualification that involves set operations between two sets. A team qualification may be of the form:

\[
tuple-set(\text{Attr}_1, \text{Attr}_2, \ldots, \text{Attr}_n) \begin{align*}
&\text{set-op} \quad \text{tuple-set(Attr}_1, \\
&\quad \text{Attr}_2, \ldots, \text{Attr}_n) \\
&\quad \text{set-op} \quad \text{tuple-const-value, or} \\
&\quad \text{aggregate-fun(tuple-set(Attr}_1, \\
&\quad \text{Attr}_2, \ldots, \text{Attr}_n)) \text{rel-op-const-value}
\end{align*}
\]

where \( \text{Attr}_1, \text{Attr}_2, \ldots, \text{Attr}_n \in W, n \geq 1 \), \( \text{tuple-set(Attr}_1, \\
\quad \text{Attr}_2, \ldots, \text{Attr}_n) \) is a set of tuples over the specified attributes \( \text{Attr}_1, \text{Attr}_2, \ldots, \text{Attr}_n \), \( \text{set-op} \) is normally one of the set operators \( \{\in, =, \subseteq, \supseteq, \neq\} \), \( \text{tuple-const-value} \) is a set of \( n \)-tuple constant values, and \( \text{aggregate-fun} \) is normally one of the aggregate functions \( \{\text{SUM, AVG, COUNT, MIN, MAX}\} \). An \( n \)-tuple constant value is a tuple which is composed of \( n \) constant values. The semantics of the well-known set operations such as \( \subseteq, \supseteq, \sim \), are the same in our definition. A bag is like a set except that its elements can be duplicate. In our definition, we particularly distinguish the difference between the definitions of the set equality \( (\in) \) and bag equality \( (\subseteq) \). Set \( A \) is equal to set \( B \) if they both have the same members. Bag \( A \) is equal to bag \( B \) if they both have the same members and the numbers of the same member in these two sets are equal.

2.2 Example Team Queries
We present two queries to illustrate various features of the language. In the following queries, underlined attributes are keys.

**Query 1.** The query uses a Baseball Database with two relations, which are given below.

\[
\begin{align*}
\text{PLAYERS(p&&r, year-of-birth, year-joined, batting-avg, team-name)} \\
\text{PLAYER-POSITION(plaver. position, #of-times)}
\end{align*}
\]

The relation \( \text{PLAYERS} \) gives the number, year-of-birth, year-joined, batting average, team-name of each player. The relation \( \text{PLAYER-POSITION} \) gives each player that plays certain positions. The \#-of-times is the number of times a player was played. Find a team of nine players and their corresponding positions that satisfies the following conditions.

(a). They were born after 1960 and joined before 1990.
(b). The number of a position which was played by a player must be greater than 50 times.
(c). One player plays only one position and one position is played by only one player.
(d). The nine players are composed of 3 players, 2 players, and 2 players. They belong to teams Elephant, Dragon, Lion, and Tiger respectively.
(e). The total batting average of the team is greater than 0.28.

The team query can be expressed as follows:

\[
\begin{align*}
\text{SELECT <TP.player, TP.position> IN TEAM <PP.player,} \\
\text{PP.position> TEAM-PLAYERS TP FROM PLAYERS P, PLAYER-POSITION PP}
\end{align*}
\]
WHERE P.player = PP.player and P.year-of-birth > 1960 and P.year-joined < 1990 and PP.#-of-times > 50 and (SELECT position FROM TP) == { pitcher, catcher, 1st, 2nd, 3rd, short, left, center, right} and (SELECT P.team-name, COUNT(P.player) FROM P GROUP BY P.team-name) == (Elephant, 3), (Dragon, 2), (Lion, 2), (Tiger, 2) and AVG (SELECT P.batting-avg FROM P, TP WHERE P.player=TP.player) > 0.28

Query 2. The query uses a Orchestra Database with six relations which are given below.

INSTRUMENT(iname, first-player)
MUSICIAN(mname, sex)
SYMPHONY(sname, time)
ORCHESTRA(sname, iname, mname)
TOUR(city, sname)
PLAYS(iname, first-player)

The relation INSTRUMENT gives the name of the first player of each instrument. MUSICIAN contains information about players. The relation SYMPHONY gives the duration in minutes of each symphony. The relation ORCHESTRA gives the set of players that execute each instrument in each symphony. TOURS contains the set of symphony names to be played in each city. The relation PLAYS gives each player that plays certain instruments. The #of-times is the number of times of an instrument that a player played. Now we want to find a group of ten players to play symphonies. It satisfies the following conditions.

(a) The ten players consist of 3 players for violin, 1 player for conductor, 2 players for cello, 2 players for viola, and 2 players for bass drum.
(b) Each player in the group must have been acted the first player of some instrument.
(c) The ten players are composed of 6 male musicians and 4 female musicians.
(d) The #of-times of an instrument which was played by a musician must be greater than 300 times.
(e) Each player must have been played the symphony "Tchaikovsky" in the city New York.

The team query can be expressed as follows:

SELECT <OT.mname, OT.iname> IN TEAM <PL.mname, PL.iname> FROM INSTRUMENT IN, MUSICIAN MU, ORCHESTRA ORCH, TOUR, PLAYS PL WHERE PL.mname = ORCH.mname and ORCH.sname = "Tchaikovsky" and ORCH.#of-times > 300 and TOUR.city = "New York" and TOUR.sname = "Tchaikovsky" and PL.mname = IN.first-player and (SELECT iname, COUNT(iname)) FROM OT GROUP BY iname == { conductor, 1), (violin, 3), (cello, 2), (viola, 2), (bass drum, 2)} and (SELECT mname, COUNT(mname)) FROM OT GROUP BY mname == { (*, 1) }

3. BASIC TEAM QUALIFICATIONS

In this section, we turn our attention to a collection of team qualifications. The team qualifications are usually divided into two groups. One group includes team qualifications from mathematical theory, which are applicable because each operand is defined to be a set of tuples. These team qualifications include BTQ1, BTQ2, BTQ3, and BTQ4. The other group consists of special team operations which are specifically defined for query optimization. These operations include ST01 and ST02, ST03, ST04, ST05, and ST06. Before starting our definition, we use the following notations:

(a) $sf(f) = SELECT <attribute-list> FROM f WHERE <search-condition>$;
(b) $sql(f, t_1, . . . , t_n) = SELECT <attribute-list> FROM f WHERE <search-condition>$;
(c) $sgf(f) = SELECT <attribute-list> FROM f WHERE <search-condition>$;
(d) $bsqlf(t) = SELECT <attribute-list> FROM t WHERE <search-condition>$;
(e) $agf1(f, t_1, . . . , t_n) = Aggregate-fun(sql(f, t_1, . . . , t_n))$ IS MAXIMUM;
(f) $agf2(f, t_1, . . . , t_n) = Aggregate-fun(sql(f, t_1, . . . , t_n))$ IS MINIMUM;
(g) $agf3(f, t_1, . . . , t_n) = Aggregate-fun(sql(f, t_1, . . . , t_n)<constant-value)$;
(h) constant-set = $(c_1, . . . , c_m)$;
(i) $tuple-constant-set = t = (c_1, . . . , c_m)$.

Where $f$ is a team name; $t$ and $t_i$, $i = 1, . . . , n$, are general table names; $s_i$, $i = 1, . . . , p$ are instances of some attribute of a relation; $n_i$, $i = 1, . . . , p$ are positive integers. A <search-condition> is a boolean expression. A <attribute-list> is a list of attributes. Aggregate-fun is normally one of the aggregate functions (SUM, AVG, COUNT, MIN, MAX). A constant-value is a constant value. A constant-set is a set of constant values. A tuple-constant-set is a set of 2-tuple constant values.

Basic Team Qualifications

(BTQ1) $sf(f) = constant-set or bsqlf(t)$
(BTQ2) $sf(f) = constant-set or bsqlf(t)$
(BTQ3) $sql(f, t_1, . . . , t_n) = Aggregate-fun(sql(f, t_1, . . . , t_n)), \theta \in \{=, \neq, <, \leq, >, \geq\}$
(BTQ4) $sgf(f) \theta tuple-constant-set, \theta \in \{=, \geq\}$

Special Team Operations

(ST01) $(constant-set or bsqlf(t))$ match (constant-set or bsqlf(t))
(ST02) $(constant-set or bsqlf(t))$ maxmatch
(ST03) $(bsqlf(t))$ mincover
(ST04) $(bsqlf(t))$ cover
(ST05) $(bsqlf(t))$ minpart
(ST06) $(bsqlf(t))$ partic

Aggregate Function

(AGF1) $agf1(f, t_1, . . . , t_n)$
(AGF2) $agf2(f, t_1, . . . , t_n)$
(AGF3) $agf3(f, t_1, . . . , t_n)$

For more detailed description about basic team qualifications and special team operations, one is referred to the contents of 13. 
4. QUERY PROCESSING

In this section we describe query processing techniques of each team qualification.

Two important things must be considered in processing a team query. One is the choice of a team candidate, and the other is the evaluation of team qualifications. In this paper, we focus on the former, that is, how to reduce the search space. In fact, many team candidates can be avoided in generating query processing. For example, consider the following team query:

\[
\text{SELECT} \quad \langle \text{TEAM-SP.s#}, \text{ TEAM-SP.p#} \rangle \quad \text{IN TEAM} \\
\langle \text{SP.s#}, \text{SP.p#} \rangle \quad \text{TEAM-SP} \\
\text{FROM SP} \\
\text{WHERE} \quad (\text{SELECT} \quad \# \text{ FROM TEAM-SP}) = \{s1, s2, s3 \}
\]

Step 1 projects \( R \) over the specified attributes. This step is used to construct the domain of a team. Step 2 selects those tuples over the specified attribute that are in the given set. This step also excludes those tuples over the specified attribute that are not in the given set. The search space can be greatly reduced in this step. In Step 4b, owing to the operation is a bag equality, thus certain tuples for each distinct instance over the specified attribute are chosen according to the number of the value in the given set. In Step 4a, one tuple or more tuples for each distinct instance is/are chosen. A team candidate is generated in this step. The team candidates then is evaluated. If all team qualifications return True, \( T \) is returned and the algorithm terminated. Otherwise goto Step 4a or Step 4b depending on BTQ1 or BTQ2. The algorithm terminates when either a team is found, or no such team exists. If all teams are required to generate, then the process is repeated until all teams are generated.

4.2 The Algorithm for BTQ4

\[(BTQ4) \quad \text{(SELECT Attr1, COUNT (Attr2) FROM T GROUP BY Attr1)} \equiv \text{tuple-constant-set} \]

Let tuple-constant-set = \{(s1, n1), (s2, n2), ..., (sp, np)\} is a set of 2-tuple constant values, where \( si, i = 1, ..., p \) are instances of some attribute of a relation, and \( n_i > 0, i = 1, ..., p \). If BTQ4 is used as a team generator, then the generator uses the following algorithm to generate team candidates.

Algorithm 2

Step 1. PROJECT (R.Attr1, R.Attr2, ...) Giving S
Step 2. SELECT (S.1 = s1), i = 1, ..., p Giving S*
Step 3. Sort the tuples in S* on attribute Attr1
Step 4. Select \( n_1 \) tuples for the instance \( s_1, n_2 \) tuples for the instance \( s_2, ..., n_p \) tuples for the instance \( s_p \) in \( S^* \) and give \( T \).
Step 4a. (match) Applying bipartite graph matching algorithm Giving S'.

Step 4b. (maxmatch) Applying maximum-weight bipartite graph matching algorithm Giving S'.

Step 5. Set-to-relation(S) Giving T.

Relation-to-Graph is used to transform relation T with two attributes into a bipartite graph G, where V1={c11, c12, ..., c1n} and V2={c21, c22, ..., c2m}, and E={(c1i, c2j), ..., 1 ≤ i, j ≤ n, 1 ≤ k ≤ p, over the attributes Attr1, Attr2, and A constructs an edge (c1i, c2j) which has a weight wk. Step 4a applies a matching algorithm. Step 4b applies a maximum-weight matching algorithm. The result of this step is a set of ordered pairs. Step 5 transforms a set of ordered pairs into a relation.

4.5 The Algorithms for ST03 and ST04

The general form of the mincover and cover operations is as follows:

\[(\text{bsqlf}) \text{mincover}\_A \text{ (constant-set or bsqlf(t))} \]

\[(\text{bsqlf}) \text{cover}\_C \text{ (constant-set or bsqlf(t))} \]

The mincover\_A operation is to find teams with minimum total value of attribute A. It may take a very long time to find a team with minimum value. The cover\_C operation is to find feasible teams whose cost satisfies certain constraints. These two operators can be modeled as set covering problems. We describe genetic algorithms to implement these operations.

Formally, let E be an \(m \times n\) binary matrix, \(w\) an \(n\)-dimensional nonnegative vector, and \(x\) an \(n\)-dimensional binary vector, with \(wx\) the inner product of the two, and \(1\) an \(m\)-dimensional column vector of ones. Then the set covering problem\(^3\) can be formulated as

\[
\begin{align*}
\text{minimize} & \quad \text{wx} \\
\text{subject to} & \quad \text{Ex} \geq 1 \\
& \quad x_i = 0, 1 \quad \text{for} \quad i = 1, ..., n
\end{align*}
\]

Any \(x\) satisfying (2) and (3) is called a cover solution.

The genetic algorithm deals with a population of chromosomes, each of which can be decoded into a solution of the problem. For each chromosome in the population, a fitness function is calculated. Chromosomes are selected from the population to become parents, based on a fitness function. Then, reproduction occurs between pairs of chromosomes to produce the offspring. The newly created population becomes the next generation and the process is repeated.

The structure of a simple algorithm is as follows:

**GENETIC ALGORITHM**

procedure genetic algorithm
begin
  t = 0
  initialize P(t)
  evaluate P(t)
  while (not termination-condition) do
    begin
t        t = t + 1
        select P(t-1) from P(t-1)
        recombine P(t)
        evaluate P(t)
    end
  end
end

4.5.1 The Algorithm for ST03

During iteration \(t\), the genetic algorithm maintains a population \(P(t)\) of solutions \(x_1, ..., x_n\) (the population size \(n\) remains fixed). Each solution, \(x_i\), is evaluated by computing \(f(x_i)\), a measure of fitness of the solution. A new population, \(P(t + 1)\), is then formed: we select solutions to reproduce on the basis of their relative fitness, and the selected solutions are recombined by use of genetic operators (such as mutation and crossover) to form the new population.

A solution of the set covering problem can be easily encoded as a binary chromosome \((0, 1, 1, 0, ...\) which would be interpreted as the second and third columns are included in the solution, but the first and fourth are not. The fitness function \(F(x)\) is a large number \(MAX\) minus the sum of the cost of the columns used (i.e., the original objective function value) and a penalty for a failure to cover. Therefore, the fitness function is

\[
F(x) = MAX - \sum_{i=1}^{n} w_i x_i - P(x)\eta(x)
\]

where \(x_i\) and \(\eta(x_i)\) are binary with \(\eta(x_i) = 0\) if the solution \(x\) is a cover and \(\eta(x_i) = 1\) otherwise. \(P(x)\) is the penalty function procedure for the covering. The latter term

\[
\sum_{i=1}^{n} w_i x_i + P(x)\eta(x)
\]

will be referred to as "cost."

For these two operations, we adopted the greedy crossover and \(P3\) in\(^{18}\). To find a cover with lower cost, we modified the penalty function \(P3\) in\(^{18}\). The greedy crossover and the penalty function \(P3\) will be explained later. We use the natural encoding/decoding of chromosomes that represent solutions of the problems together with associated greedy crossover and new penalty function to implement the cover and mincover (sumcover) operations.

The penalty function procedure \(P3\) of\(^{18}\) is stated as follows:

P3: If \(S\) is a cover, cost = \(\sum W_i\)

If \(S\) fails to be a cover, let \(R\) be the rows that remain uncovered. For each \(i\) in \(R\), let \(S_i\) be the columns that cover \(i\). Let \(w^*_i = \min\{w_j\}\) for \(j\) in \(S_i\). Then cost = \(\sum_{i=1}^{n} W_{j*}\) + \(\sum_{i \in R} W_i\). In order to find a cover with lower cost, the above penalty function procedure \(P3\) is simply modified to become \(P3'\) as follows:

P3': If \(S\) is a cover, cost = \(\sum W_i\)

If \(S\) fails to be a cover, let \(R\) be the rows that remain uncovered. For each \(i\) in \(S\), strike column \(j\) and the rows covered by \(j\). For some \(i\) in \(R\), let \(S_i\) be the unused columns that cover \(i\). For each \(j\) in \(S_i\), calculate the cost-ratio\([j] w_j / \text{number-of-1's-uncovered in } j\). Let \(j^* = \min\{\text{cost-ratio}(j)\}\) for \(j\) in \(S_i\), Append to \(S\) the column \(j^*\).

The process is repeated until \(S\) becomes a cover.

Where cost-ratio\([j]\) is the cost of column \(j\) over the number of 1's in the column \(j\).

The greedy crossover\(^{18}\) is described as follows:

For every pair of parent genes:

Step 1. Initialize the set \(S\) to be empty and the matrix \(A\) to
be the original set covering matrix with column costs (\(w_j\)).

Step 2. For the unused columns and uncovered rows, calculate the cost-ratio (\(w_j / \text{number-of-1's-uncovered}\)).

Step 3. Append to \(S\) the column (say column \(c\)) with the least cost-ratio that is included in one of the parents. (Break ties randomly.)

Step 4. Strike column \(c\) and the rows covered by \(c\) from \(A\) and let this new matrix be \(A'\).

Step 5. If \(S\) is a cover or if no other columns are represented in the parents, stop. Otherwise, set \(A\) to \(A'\) and go to Step 2.

We use simple genetic algorithm with random initial population, proportional selection, generational replacement, greedy crossover and no mutation for these two operations. In the calculation of fitness penalty function procedure \(P3\) is used for cover\(C\) operations, while \(P3'\) is used for mincover\(A\) operation.

### 4.6 The Algorithms for ST05 and ST06

The minpart\(A\) and part\(C\) operations can be modeled as set partitioning problems. Similarly, a solution of the set partitioning problem can be easily encoded as a chromosome. In this subsection we use the greedy crossover of\(^{18}\) and penalty function procedure \(PP\) to implement the operations.

The genetic algorithm starts by checking whether an individual violates the definition of partition. That is to say, no element is covered by more than one set. If an individual violates the definition of set partitioning problem, then a mutation operator is repeatedly applied until no individual violates the definition of set partitioning problem. Then the algorithm computes the fitness of all valid individuals by using the given penalty function procedure \(PP\) and uses the greedy crossover of\(^{18}\) to generate the offspring. Finally, the algorithm examines whether the partition solution covers all elements. If the partition solution does not cover all elements, then it attempts to improve the partition solution obtained. The penalty function procedure \(PP\) is described below:

\[
PP: \text{if } S \text{ is a partition, cost } = \sum_{i \in S} w_i
\]

If \(S\) fails to be a partition, let \(R\) be the rows that remain uncovered. For each \(j \in S\), strike column \(j\) and the rows covered by \(j\). In addition, for each \(j \in S\), every column \(k \neq j\) such that \(a_{ij}=a_{ik}=1\) \((i=1, \ldots, m)\) must be deleted. For some \(i \in R\), let \(S_i\) be the unused column that cover \(i\). If \(S_i\) is an empty set, let cost \(= \infty\). Otherwise, for each \(j \in S_i\), calculate the cost-ratio \((w_j / \text{number-of-1's-uncovered in } j)\). Let \(j^* = \min\{\text{cost-ratio}(j)\}\) for \(j \in S_i\). Append to \(S\) the column \(j^*\). The process is repeated until \(S\) is a partition or the cost is \(\infty\).

The algorithm for minpart\(A\) and part\(C\) is described below.

**Step 1.** If any row \(r\) of \(A\) has all 0's, there is no solution since constraint \(r\) cannot be satisfied.

**Step 2.** Choose a desired population size \(n\) and initialize the starting population \(P\) randomly.

**Step 3.** Check whether a chromosome of the population \(P\) violates the definition of set partitioning problem. If any chromosome violates the definition, mutate a randomly chosen bit of 1's of this chromosome. The step is repeated until no chromosomes of population \(P\) violate the definition of set partitioning problem.

**Step 4.** Evaluate fitness of each individual according to the penalty function procedure \(PP\).

**Step 5.** If the fitness of all individuals is equal to 0, there is no partition solution. The algorithm returns "No Partition, Solution" and terminates.

**Step 6.** If termination-condition held, go to Step 7; else probabilistically select a pair of individuals (according to their fitness determined by the penalty function procedure \(PP\)) to generate the offspring using greedy crossover. The step is repeated until \(n\) offspring are generated. Return to Step 3.

**Step 7.** If the resultant solution, say \(S\), contains a partition solution, the algorithm terminates. If \(S\) fails to be a partition, let \(R\) be the rows that remain uncovered. For each \(j \in S\), strike column \(j\) and the rows covered by \(j\). In addition, for each \(j \in S\), every column \(k \neq j\) such that \(a_{ij}=a_{ik}=1\) \((i=1, \ldots, m)\) must be deleted. For some \(i \in R\), let \(S_i\) be the unused columns that cover \(i\). If \(S_i\) contains only one column, append to \(S\) the column \(c\) (say column \(c\)). Strike column \(c\) and the rows covered by \(c\). Otherwise, for each \(j \in S_i\), calculate the cost-ratio \((w_j / \text{number-of-1's-uncovered in } j)\). Let \(j^* = \min\{\text{cost-ratio}(j)\}\) for \(j \in S_i\). Append to \(S\) the column \(j^*\). The step is repeated until \(S\) is a partition.

The reason why we need Step 7 is that the genetic algorithm sometimes found partial solutions with few elements not covered. Therefore, Step 7 in the algorithm attempts to improve the partition solution obtained in Step 6 by applying the procedure like the penalty function procedure. After applying this local fix-up procedure of Step 7, feasible solutions have been always produced.

### 5 PROPERTIES

There are many properties that can be used for transforming a sequence of team qualifications into equivalent ones. Then an optimization algorithm can utilize some of these properties to transform an initial query into an optimized form that is more efficient to execute (in most cases). In this section, we state some important properties without proving them. The following notation is used in their description: the symbol \(BTQ_i\), \(i=1, 2, 3, 4, j=1, 2, \ldots, 14\), represents the \(j\)th team qualification of \(BTQ_i\). For example, \(BTQ_21\) represents the first team qualification of \(BTQ_2\).

**Definition.** Two team expressions are equivalent if the sets of all the possible teams of those expressions are identical.

**Property 1.** Commutativity of \(BTQ_i\):

\[(BTQ_1 \land BTQ_2) = (BTQ_2 \land BTQ_1)\]

**Property 2.** Commutativity of \(BTQ_i\):

\[(BTQ_1 \land BTQ_2) = (BTQ_2 \land BTQ_1)\]

**Property 3.** Suppose a team is composed by a set of tuples over attributes \(A_1\) and \(A_2\) in relation \(R\). Let \(BTQ_i\) be a team generator and \(BTQ_i\) be one of other team qualifications. The expression \((BTQ_1 \land BTQ_4)\) can be transformed into a \(BTQ_2\) if they satisfy certain constraints.

\[(BTQ_1 \land BTQ_4) \Rightarrow BTQ_2\]

**Property 4.** Suppose a team is composed by a set of tuples over attributes \(A_1\) and \(A_2\) in relation \(R\). Let \(BTQ_i\) be a team generator, and \(BTQ_i\) be one of other team qualifications. The expression \((BTQ_2 \land BTQ_4)\) can be transformed into a special team operation \((STO_i)\) if they satisfy certain constraints. That is,

\[(RTQ_2 \land RTQ_4) \Rightarrow STO_1\]

250
Property 5. (i) Assume that the <attribute-list> of BTQ2 involves only one attribute, and also that the <attribute-list> of BTQ2 involves only the other attribute. Two BTQ2's can be transformed into a ST01. That is, (BTQ21 and BTQ22) \( \Rightarrow \) ST01

(ii) Assume that the <attribute-list> of BTQ4 involves an attribute, say A1, which is also used by the GROUP BY clause, and also that the <attribute-list> of BTQ4 involves the other attribute, say A2, which is also used by the GROUP BY clause. If the tuple-constant-set of two BTQ4 is of the form (*, 1), then (BTQ41 and BTQ42) \( \Rightarrow \) ST01.

Property 6. Suppose attributes X, Y, and Z are defined on relation R and attributes X and Y uniquely identify attribute Z. Suppose a team is composed by a set of tuples over attributes A1 and A2 of relation R and the <attribute-list> of AGF1 involves only Z. If we add the aggregate function AGF1 to the expressions of Properties 4 and 5, then these expressions can be combined into a special team operation ST02. That is, (BTQ2 and BTQ2 and AGF1) \( \Rightarrow \) ST02

(BTQ2 and BTQ4 and AGF1) \( \Rightarrow \) ST02

(BTQ41 and BTQ42 and AGF1) \( \Rightarrow \) ST02

Property 7. Suppose a team consists of a set of tuples over attribute A1 of R. Suppose the <attribute-list> of BTQ3 involves only attribute A2 of R and the operator between two sets is a set equality. A BTQ3 is followed by a AGF2 and a BTQ3 is followed by a AGF3 can be transformed into a ST03 and ST04 respectively. That is, (BTQ3 and AGF2) \( \Rightarrow \) ST03

(BTQ3 and AGF3) \( \Rightarrow \) ST04

Property 8. All assumptions are the same as those of Property 7 except the operator between two sets in BTQ3 is a bag equality. A BTQ3 is followed by a AGF2 and a BTQ3 is followed by a AGF3 can be transformed into a ST05 and ST06 respectively. That is, (BTQ3 and AGF2) \( \Rightarrow \) ST05

(BTQ3 and AGF3) \( \Rightarrow \) ST06

6 ANALYSIS OF EVALUATION OF TEAM QUERIES

6.1 Genetic Algorithms for mincover, cover, minpart, and part Operations

Simulations have been performed for the set covering problem on 12 test problems, and for the set partitioning problem on 10 test problems. The problem size of set covering problems was obtained from the work of Yang). The problem size of set partitioning problems was obtained from the work of Lin, Kao, and Hsulg). The reaming problem sizes were designed by the authors. The examined instances have been generated as follows. The A matrix for all set covering and set partitioning problems was obtained from a matrix generator. All the test problems have coefficient matrices whose density varies from 1% to 20%. If a row or a column of A does not contain 1 in the row or the column then one 1 is added in any entry of the row or column. That is, each row or column must at least contain one 1. Following the work of Yang), the coefficient of the objective function was equal to the number of ones in the corresponding column plus a random variable between 0 and 1. Information on these test problems, as well as on the computational results, is presented in Tables I and II. The genetic algorithms were programmed in C and run on a DEC station 5000/200.

<table>
<thead>
<tr>
<th>Table I</th>
<th>No. of test</th>
<th>m</th>
<th>n</th>
<th>Popsize</th>
<th>CPU time**</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>mincover and cover</td>
<td>1</td>
<td>15</td>
<td>16</td>
<td>0.050</td>
<td>22.132</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>30</td>
<td>16</td>
<td>0.050</td>
<td>40.984</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>200</td>
<td>64</td>
<td>17.281</td>
<td>375.320</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>200</td>
<td>72</td>
<td>24.580</td>
<td>367.645</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>50</td>
<td>60</td>
<td>4.400</td>
<td>66.269</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>50</td>
<td>60</td>
<td>3.705</td>
<td>41.410</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>104</td>
<td>64</td>
<td>10.731</td>
<td>161.914</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>700</td>
<td>80</td>
<td>23.350</td>
<td>306.076</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>46</td>
<td>60</td>
<td>7.940</td>
<td>54.867</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>26</td>
<td>60</td>
<td>4.890</td>
<td>28.910</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>50</td>
<td>64</td>
<td>12.641</td>
<td>58.920</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>134</td>
<td>120</td>
<td>62.554</td>
<td>211.726</td>
<td></td>
</tr>
<tr>
<td>Note:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>m=Number of constraints, n=Number of variables</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>*Population Size, **(seconds) DEC station 5000/200</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table II</th>
<th>No. of test</th>
<th>m</th>
<th>n</th>
<th>Popsize</th>
<th>CPU time**</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>minpart and part</td>
<td>1</td>
<td>13</td>
<td>16</td>
<td>0.269</td>
<td>15.854</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>13</td>
<td>16</td>
<td>0.191</td>
<td>16.755</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>14</td>
<td>16</td>
<td>0.246</td>
<td>17.848</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>12</td>
<td>16</td>
<td>0.292</td>
<td>15.894</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>13</td>
<td>16</td>
<td>0.312</td>
<td>13.879</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>50</td>
<td>48</td>
<td>20.049</td>
<td>62.940</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>100</td>
<td>80</td>
<td>159.875</td>
<td>129.429</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>150</td>
<td>80</td>
<td>696.775</td>
<td>183.920</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>100</td>
<td>120</td>
<td>476.942</td>
<td>113.852</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>200</td>
<td>150</td>
<td>1996.273</td>
<td>242.022</td>
<td></td>
</tr>
<tr>
<td>Note:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>m=Number of constraints, n=Number of variables</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><em>,</em>* have the same meaning as in Table I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The results of these tests are very interesting. For all these test problems, feasible solutions have been always produced. The results show that the CPU time is nearly proportional to the product of the number of constraints and the number of variables for the set covering problems. The CPU time for set partitioning problems is larger than that for the set covering problems. This is to be expected since the penalty function of set partitioning problems is much more complicated than that of set covering problems. Also, the CPU time for set partitioning problems is not linear to the product of the number of constraints and the number of variables. We are currently working on the problem of reducing the processing time of the set partitioning problems, possibly by adopting the Annealing Genetic algorithm of Lin, Kao, and Hsu(9).
Two team qualifications in a query can be transformed into a special team operation STO1 if they satisfy certain constraints. The corresponding team query is shown below. 

```sql
SELECT <T'.player, T'.position> IN TEAM <player, position> TEAM-PLAYERS T
FROM PLAYERS
WHERE ((pitcher, catcher, 1st, 2nd, 3rd, short, left, center, right) match ( SELECT player FROM PLAYERS ))
```

<table>
<thead>
<tr>
<th>Player</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>pitcher</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>catcher</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1st</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>2nd</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>short</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>3rd</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>left</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>center</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>right</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

| # of tuples in PLAYERS | 15 | 27 | 40 | 50 |

Information about the number of tuples in relation PLAYERS, the number of distinct instances of the attribute player in PLAYERS, and the number for each instance of the attribute position appears in relation PLAYERS is shown in Table III. The average of the number of tests to generate a team for different algorithms is shown in Table IV. It can directly be seen that STO1 is highly efficient than other two algorithms. All algorithms were programmed in C and run on a 80486-based PC.

### REFERENCES


