Genetic Algorithms for Large Join Query Optimization

Hongbin Dong, Ph.D.
International School of Software,Wuhan University
Wuhan, Hubei, P.R.China 430072
86 - 27 - 68778870
hbdong@whu.edu.cn

Yiwen Liang, Ph.D.
Computer Science School, Wuhan University
Wuhan, Hubei, P.R. China 430072
86 - 27 - 61080011
ywliang@whu.edu.cn

ABSTRACT
Genetic algorithms (GAs) have long been used for large join query optimization (LJQO). Previous work takes all queries as based on one granularity to optimize GAs and compares their efficiency with other query optimization algorithms. However, we believe that large join queries are based on a granularity that is too large (1) to optimize GAs and (2) to compare the efficiency of different randomized optimization algorithms. Besides, while previous work only discusses the efficiency of basic GAs for LJQO, we believe that hybrid GAs reduce search space to improve GAs efficiency.

We will present a genetic optimization model which includes factors affecting the efficiency of GAs. In this model, the query model is the granularity upon which GAs are optimized. Based on six typical query models, experiments have been done, first, to optimize four classes of GAs; and second, to prove the rationality of the query model as a trade-off between the efficiency and robustness of GAs. Finally, we will provide suggestions for choosing one of four classes of GAs and for the settings and combinations of components of GAs.

Categories and Subject Descriptors
H.2.4 [**Database Management**]: Systems—*query processing, relational database*

General Terms
Algorithms, Performance, Experimentation

Keywords
genetic algorithm, large join query, optimization, query model

1. INTRODUCTION
In the relational database systems the join is one of the relational algebraic operators and has a very high execution cost. Join query optimization transforms a declarative query expression into procedural query execution plans (QEPs) and determines the lowest query execution plan (QEP). But join query optimization is a combinatorial optimization problem. When a query includes more than 10 relations (this is called a large join query), the exhaustive optimization algorithms can not optimize the query in a reasonable period of time due to their time complexity.

However, many new relational database applications produce very complex join queries. A very large database system is one kind of application. The decision support system is another. The data warehouse is the data center of a decision support system and uses a relational model to store and manage data. The data flowing in (data are integrated and transferred from multiple databases and flat files to a data warehouse) and out (data in the data warehouse are transferred to data for on-line analytical processing tools and data mining tools) of the data warehouse produce very large join queries.

To optimize large join queries, randomized local search algorithms, such as Simulated Annealing (SA) and Iterative Improvement (II), have been used as viable alternatives to exhaustive searches. Based on one regular query model, Swami and Gupta test SA and II, then conclude that the relatively simple II is better than all other local search algorithms including SA [12]. The interesting thing is that a different conclusion arises later. Based on three star schema query models, Ioannidis tests SA and II, then concludes that in most cases SA identifies a lower cost QEP than II [7].

While randomized local search algorithms have proved their efficiency for large join queries, they have been found to be easily trapped in local optima as well. Due to the nature of global searching and their successful application to different combinatorial optimization problems, genetic algorithms (GAs) are used to solve large join query optimization (LJQO). Bennet et al. start to design GAs and compare GAs with the System-R algorithm for queries with a maximum 16 joins and the experiments show that in left-deep tree space GAs can find near optimal QEP, but the optimizing time is longer than the System-R algorithm, while in the whole space, GA can find a better QEP than System-R's [2]. Steinbrunn et al. compare the efficiency of heuristic, randomized local search algorithms and GAs for queries with a maximum 30 joins. It concludes that a heuristic avoids high time complexity but the optimized result is rarely acceptable, while randomized and genetic algorithms need a longer optimizing time, but the optimized result is far better [10].

Lahiri and Kang compare a GA in left-deep tree space and a genetic programming (GP) in the whole space, and conclude that the GP performs significantly more effectively than the GA [8]. Stillger and Spiliopouhou propose a GP, compare the GP with II, and conclude from the experiments of using two cost models that the GP can converge towards the near optimal QEPs but II is more effective [11].

Previous work in randomized optimization algorithms for large join query optimization is based on all queries to optimize algorithms and compare their efficiency. In fact, they are based on an assumption that some randomized algorithms are generally better than others for any queries. However, for two reasons we think all queries are based on too large a granularity due to the wide range of complexity and diversity of queries. First, GA is a randomized algorithm and its efficiency depends on the nature of the problem. The settings and combinations of the components of GAs decide the efficiency of a GA for the problem. Second, according to the no-free-lunch theorem, no algorithm can solve all problems and be generally superior to all competitors [9].

In this paper, we will explore a proper granularity, a good trade-off between the efficiency of GAs and robustness of GAs, then explain the rationality of this granularity. Besides the basic GAs used by previous work for LJQO, we will also test the efficiency of different hybrid GAs, then compare the efficiency of different GAs.

2. LJQO PROBLEM

The optimizing process for a large join query includes two parts: logical optimization and physical optimization. The query, submitted in high-level declarative language such as SQL, is translated to a query graph as the input of a logical optimizer. In a query graph each node represents a base relation and each edge a common attribute between two connected base relations. According to relational algebraic equivalent rules — the associative rule and commutative rule of join operators — the logical optimizer generates many equivalent join trees for the query graph as its output. In a join tree each leaf represents a base relation and each inner node a join operator. A join operator is the join result of its left child and right child. For each join tree, a physical optimizer selects a physical operator from several candidates for any join operator in the join tree to produce many equivalent operator trees. In an operator tree each leaf represents a base relation and the inner node is a physical operator. A physical operator is an algorithm for executing the join operator. Finally, the optimizer estimates the cost of each operator tree then determines the lowest cost operator tree, the optimal QEP.

Therefore, the optimizing process is meant to choose the optimal join order for the relations in a query graph then to determine the optimal join algorithm for each join operator. We will focus on the logical optimization, i.e. choose an optimal join order for a query. The logical optimization is still a combinatorial optimization problem.

The solution space of a query is the set of all QEPs for the query. The goal of large join query optimization is to find the optimal QEP in the solution space. A desirable LJQO will solve three sub-problems: (1) choose a search space which is the subset of the solution space and includes lowest cost QEP, (2) choose a cost model which can accurately estimate the cost of QEPs, and (3) choose a search algorithm which is effective and efficient. Each of these three tasks is nontrivial.

Commonly, according to the shape feature of join trees, a solution space is divided to three non-overlapping search spaces: a left-deep tree space, a right-deep tree space, and a bushy space. If any inner node has a base relation as right child, the tree is called a left-deep tree. If any inner node has a base relation as left child, the tree is called a right-deep tree, which is symmetrical to a left-deep tree. Otherwise, it is called bushy. Given N is the number of relations in a query, the left-deep tree space has N! QEPs, right-deep tree space has N! QEPs and the whole solution space has $C_{2N-2}^{N-1} * (N-1)!$ QEPs. As is the case for most LJQO algorithms, we use left-deep tree space as the search space.

A cost model is a set of formulas that is used to estimate the cost of a QEP before it is run. Generally the cost includes the CPU cost and disk access cost. In a large database system the disk access cost is much higher than the CPU cost, so we take into account only the disk access cost. We use the simple cost model used in [8]. The GAs we designed can be easily changed to other cost models.

The simple cost model is based on two assumptions: (1) the uniform distribution of attribute values and (2) the sum of the size of the intermediate relations determines the cost of a join tree. The formulas are as follows:

If t_i (i = 1,2,...,n-1) is an inner node in a join tree, then the cost formula of the join tree is

$$Cost = \sum_{i=1}^{n-1} n(t_i) \tag{1}$$

Where, $n(t_i)$ is the number of tuples in the relation t_i

For inner node t, if r and s are relations represented respectively by left child and right child of t, and C is a common attribute group in relation r and s, then:

$$n(t) = \frac{n(r) \times n(s)}{\prod\limits_{Cj \in C} \max(V(Cj, r), V(Cj, s))} \tag{2}$$

$$V(A,t) = \begin{cases} V(A,r) & A \in r - s \\ V(A,s) & A \in s - r \\ \min(V(A,r), V(A,s)) & A \in r \& A \in S \end{cases} \tag{3}$$

V(A,r) is the number of distinct values that appear in the relation r for attribute A.

To estimate the cost of a join tree, formulas (2) and (3) are used for every inner node (or intermediate relation) to calculate the number of tuples (or size) and the number of distinct values for its join attributes. Then formula (1) is used to sum up the size of every inner node. So the cost estimating of a join tree consumes much computation time.

Generally, search algorithms are classified into three categories: (1) Exhaustive algorithms; (2) Heuristic algorithms; (3) Random search algorithms. GAs are a class of random search algorithms.

3. A GENETIC OPTIMIZATION MODEL

3.1 Four types of GAs for LJQO

Basic GAs include the following components: (1) Code; (2) Fitness function; (3) Genetic factors, which include initialization operator, crossover operator, mutation operator, selection strategy, replacement strategy, and termination criteria; (4) Genetic parameters: population size (N), crossover probability (P_c) and mutation probability (P_m). The settings and combinations of all components decide the efficiency of GAs. So, finding an efficient GA for the problem itself is an optimization problem.

Hybrid GAs combine some problem-specific heuristics, or constraints, into genetic operators, or add extra steps of randomized local search methods into basic GAs to enhance the performance. Generally, the hybrid GAs include constraint GAs, heuristic GAs, and genetic local search (GLS) algorithms. Constraint GAs improve performance by reducing a search space through problem-specific constraints. Heuristic GAs use problem-specific heuristics to reduce a search space. GLS algorithms are based on a basic GA framework but perform local search algorithms for each individual before genetic operators [6]. In fact, GLS lets GA search the local optima's space instead of the whole search space. The goal of this paper is to test the existence of variation on efficiency of four types of GAs for LJQO. We are not going to test all constraints or all heuristics or all local search methods in relational database systems to determine the optimal GAs. Instead, we use the commonly used constraint – avoiding Cartesian-products to design constraint GAs, use the smallest cardinality heuristic to design heuristic GAs, use two commonly used local search methods – Swap and 3Cycle, which are proposed in [12], to design GLS algorithms.

3.2 A Query Model as a Granularity

The efficiency of GAs depends on the nature of the problem. That means the GAs for LJQO have to be optimized based on join queries. Because the efficiency and robustness are two conflicting goals, the granularity of join queries has to be chosen carefully. Too large a granularity will produce very high robustness but low efficiency, while too small a granularity will produce low robustness but high efficiency. So, a proper granularity has to be chosen carefully.

A query model is the method of characterizing a query graph. Factors, such as number of relations, shape of query graph, distribution of relation cardinalities, distribution of distinct values of attributes, and join cutoff probability, characterize a query model.

We believe that a query model will be a proper granularity to be based on to optimize GAs. From the application aspect, an assumption has existed that the queries in a database application or a data warehouse application follow a query model. Therefore, first, we can find the query model through sampling queries from the application environment, and then optimize GAs based on this query model to determine the optimal GA. When a query is submitted to the database system, the optimal GA is used to optimize this query. The advantage is that the process of optimizing GA is offline. Even more, when the application environment has been changed too much to fit the original query model, the system can go through sampling queries, to determine the new query model, and then optimize GA again.

3.3 A Genetic Optimization Model for LJQO

Taking left-deep tree space as a search space and the simple cost model [8] as a cost model, the task of GAs for LJQO is to determine the optimal GA. The efficiency of GAs for large join query optimization is affected by many factors. Based on a query model to optimize GAs, we present a genetic optimization model for LJQO as follows:

Opti_model (Query_model, GA_model)
Query_model(Num_rel, GraphJoin_ProR_disA_dis)
GA_model (Code, Func, Init, Cros, Mut, Selec, Repla, Stop crit, N, P_c, P_m)

Where Opti_model is the genetic optimization model for LJQO and its elements are two sub-models: Query_model, and GA_model. Query_model is a query model. GA_model is a genetic algorithm model. The goal of Opti_model is based on queries from a certain Query_model to determine an optimal GA determined by a GA_model.

The elements of a Query_model(Num_rel, Graph, Join_Pro, R_dis, A_dis) are respectively mapped to all factors of a query model as follows:number of relations, shape of query graph, distribution of relation cardinalities, distribution of distinct values of attributes, and join cutoff probability.

The elements of a GA_model (Code, Func, Init, Cros, Mut, Selec, Repla, Stop crit, N, P_c, P_m) are respectively mapped to all components of GAs as follows: code, fitness function, initialization operator, crossover operator, mutation operator, selection strategy, replacement strategy, termination criteria, population size, crossover probability, and mutation probability.

3.4 Experimental Methods and Evaluation

To optimize GAs based on the query model, we need to generate a large number of queries (from 10 to 100) from different query models. We choose six query models: three typical query models G1, G2, G3 in databases and three typical query models ST, SN, MS in data warehouses(DWs), see Table 1.

G1 is a query model that random query optimization algorithms commonly use as a benchmark [12]. G2 is a variation of G1 that makes uniform the distribution of the relation cardinalities. G3 is a variation of G1 that doubles the join_probability. ST is a query model in which the shape of the query graph is a star. SN is a variation of ST in which the shape of the query graph is a snowflake. MS is a variation of ST in which the shape of the query graph is a multi-star. For each query model, ten query examples are generated randomly. Their sizes range from 10 to 100 in increments of 10. For the query model X (here X is G1, or G2, or G3, or ST, or SN, or MS), the ten query examples are XQ01, XQ02, XQ03, XQ04, XQ05, XQ06, XQ07, XQ08, XQ09, and XQ10. For each query, each GA is run ten times and the averages are evaluated. To evaluate the efficiency of GAs we compare the optimizing time and optimal QEP cost because the response time for a query includes two parts: the time of a GA optimizing a query to find an

Table 1: Typical query models in DBs and DWs

model	Join_Pro	R_dis	A_dis
G1	0.01	[10, 10^2] — 20% (10^2, 10^3]— 60% (10^3, 10^4— 20%	(0, 0.2] — 90% (0.2, 1)— 9% 1.0 — 1%
G2	0.01	[10, 10^4] — 100%	(0, 0.1] — 90% (0.1, 1)— 9% 1.0 — 1%
G3	0.02	[10, 10^2] — 20% (10^2, 10^3]— 60% (10^3, 10^4)— 20%	(0, 0.2] — 90% (0.2, 1)— 9% 1.0 — 1%
ST		[10^5, 10^6]—center point [10^2, 10^4)— 100%	(0, 0.2] — 10% (0.2, 1)— 90%
SN		[10^5, 10^6]—center point [10^2, 10^3)— 100%	(0, 0.2] — 10% (0.2, 1)— 90%
MS		[10^5, 10^6]—center point [10^2, 10^4)— 100%	(0, 0.2] — 10% (0.2, 1)— 90%

optimal QEP and the time of executing the optimal QEP. We use the ratio of the optimizing time over the minimum optimizing time for the query among all GAs to measure the optimizing time, and the ratio of the optimal QEP cost over the minimum optimal QEP cost found for the query among all GAs to measure the optimal QEP cost.

4. THE PERFORMANCE OF GAS FOR SIX QUERY MODELS

Taking left-deep tree space as a search space and the simple cost model as a cost model, in this section we will design and optimize four classes of GAs for LJQO and compare their efficiency for six query models. In Sections 4.1, 4.2, 4.3, 4.4, and 4.5 our experimental results and analyses are based on the query model G1. In Section 4.6, we will show the experimental results of other query models.

4.1 Basic Genetic Algorithms

For all GAs, based on features of LJQO, we make the following restrictions to some parameters of the GA_model:

1. Code
Both the locations of relations and the adjacent relationships between relations affect the cost of QEP, so we use an ordered string of vertices to represent a QEP. Giving each basic relation a serial number started from 1, the chromosome is generated by putting the serial numbers of all leaf nodes in a left-deep tree from left to right. The advantage of the code is that its genotype is its phenotype. So it is easy to use constraints and heuristics in relational database systems into the GAs.

2. Func
The formulas of a simple cost model [8] are used as the fitness functions.

3. N
Because the computation cost of the fitness function is very high, the population size N should be small.

4. Repla
Two common replacement strategies are used: keeping diversity — replacing the worst duplicate or the worst individual if there is not a duplicate, and replacing worst — replacing the worst individual.

5. Selec
We use the random selection to maintain the diversity of population because the population size is small. But the replacement strategies we use will avoid the degeneration of the next generation population.

6. Stop crit
No improvement for 500 generations.

For the left parameters, we use the following common settings of Init, Cros, and Mut. Their details can be found in [1] [4].

a). Init
Random initialization (RI)

b). Cros
Partially-mapped crossover (PMX)
Order crossover (OX)
Uniform order-based crossover (UX)

c). Mut
Swap (2D): randomly select and exchange two points.
Flip (2F): randomly select two points and exchange two segments.
Near (2N): randomly select two points and put second point to the front of first.
3 Cycle (3D): randomly select three points and circularly move three points.

The experimental process of optimizing basic GAs is divided into three steps: (1) Determining effective crossover operators and mutation operators; (2) Determining the effective combinations of crossover and mutation operators; (3) Determining optimal settings of Repla, N, P_c, and P_m for the effective combination. The experiments show that the optimal basic GA settings are: N=60, Pc=0.75, Pm=0.25, Repla=keeping diversity, Cros=UX, Mut=2D.

4.2 Constraint Genetic Algorithms

Avoiding Cartesian-products is an efficient constraint in relational database systems to reduce a search space. The constraint is used in GAs through genetic operators: initialization, crossover and mutation. We design a constraint initialization operator – avoiding Cartesian-products initialization (ACI), a constraint crossover operator – IPPX, and a constraint mutation operator – 1D.

A constraint initialization operator – ACI

The input is a query graph with m vertices (relations) and the output is a population with n chromosomes.

```
begin
  for I = 1 to n do
```

```
    { S = NULL; num = 1;
        randomly select a relation r from the query graph
          as the num-th gene of chromosome;
      repeat
        put all relations linked to the relation r in the query
          graph but not in chromosome, into S;
        randomly select a relation r from the query graph
          as the num-th gene of chromosome;
        num = num + 1;
      until num = m; }
  end.
```

A constraint crossover operator – IPPX

The Precedence Preservative Crossover(PPX) is presented by [3]. First the offspring chromosome is initialized empty. Then a vector of length n is randomly filled with elements of the set {1, 2}. This vector defines the order in which genes are drawn from parent 1 and parent 2 respectively. After a gene is drawn from one parent and deleted in the other, it is appended to the offspring chromosome. This step is repeated until both parent chromosomes are empty and the offspring contains all genes involved.

Apparently, The Precedence Preservative Crossover(PPX) respects the absolute order of genes in parental chromosomes, but the offspring cannot avoid Cartesian-products.The Cartesian-products may appear between the first pair of adjacent genes which are from different parents.

We present the IPPX as a variation of PPX, which can avoid Cartesian-products. First, for the first gene x of parent 2, locate x in the parent 1 as i-th gene. Then, copy the 1, 2, . . . , i genes of parent 2 to the 1, 2, . . . , i genes of offspring. Finally, produce the left genes of offspring by PPX operator.

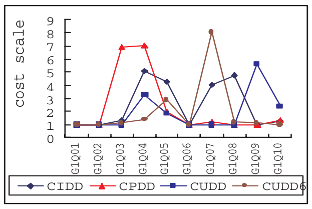
A constraint mutation operator – 1D

In a chromosome p, randomly select a relation r which is linked to the first gene in the query graph, then put r in front of the first gene. This mutation operator, we call 1D, can avoid Cartesian-products.

Table 2: Constraint GAs

CGA	N	Init	Repla	Cros	Mut
CUDD	30	ACI	Keeping diversity	UX	2D
CPDD	30	ACI	Keeping diversity	PPX	2D
CIDD	30	ACI	Keeping diversity	IPPX	2D
CUDD6	60	ACI	Keeping diversity	PPX	2D

To test the efficient combinations of constraint genetic operators and non-constraint operators, we set components of



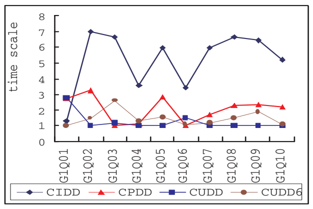


Figure 1: Costs by CGAs **Figure 2: Times of CGAs**

constraint GAs as following options: (1) Init: ACI; (2) Repla: keeping diversity and replacing worst; (3) Cros: UX, PPX and IPPX; (4) Mut: 2D and 1D. The parameters P_c and P_m are 0.75 and 0.25 which have been determined in basic GAs experiments. Experimental data show that the replacement strategy of keeping diversity is more efficient than replacing worst. An important experimental result is that GAs with N=30 have a shorter optimizing time than GAs with N=60, but their optimal GAs are very close. We believe this is because the constraint reduces the search space. Table 2 is four typical GAs with Repla = keep diversity, but different crossover, mutation operators and population size. Figure 1 is the costs of their optimal QEPs and Figure 2 is their optimizing time.

The above experimental data show:

(1) Using ACI, IPPX, and ID simultaneously damages the diversity of population and leads to premature convergence;

(2) For the crossover operators, generally, UX is better than PPX, and PPX is better than IPPX. We believe that is because IPPX excessively keeps the orders of genes of parents, which causes offspring to be too close to their parents in the search space.

(3) CUDD has the shortest optimizing time and near optimal QEPs. CUDD is the best constraint GA.

4.3 Heuristic Genetic Algorithms

We choose a common heuristic — smallest cardinality criterion, to initialize the population.

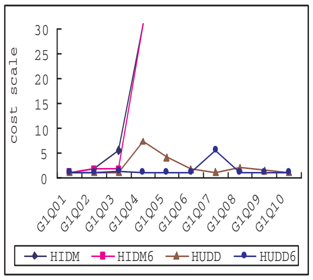
A smallest cardinality initialization (SCI)

The smallest cardinality criterion generates an individual as follow: given a set of relations S, the first relation is randomly selected from S and then repeatedly selects the relation with the smallest cardinality and avoids Cartesian-products from S until S is empty.

Table 3: Heuristic GAs

HGA	N	Init	Repla	Cros	Mut
HIDM	30	SCI	Keeping diversity	IPPX	1D
HIDM6	60	SCI	Keeping diversity	IPPX	1D
HUDD	30	SCI	Keeping diversity	UX	2D
HUDD6	60	SCI	Keeping diversity	UX	2D

We still choose the parameters P_c and P_m as 0.75 and 0.25. Other components of heuristic GAs have the following options: (1) Init: SCI; (2) Cros: UX and IPPX; (3) Mut:



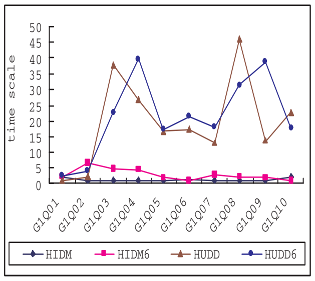


Figure 3: Costs by HGAs **Figure 4: Times of HGAs**

2D and 1D. Table 3 is four typical GAs with Repla = keep diversity, but with different crossover operators, mutation operators and population size. Figure 3 is the costs of their optimal QEPs and Figure 4 is their optimizing times.

HUDD and HUDD6 are similar and good, but HIDM and HIDM6 prematurely converge. The reason is that SCI limits the individuals of population to a very small search space and IPPX which preserves the gene orders of parents further limits the search space. Therefore, the crossover operators and mutation operators, which preserve genes orders of parental chromosomes, should not be used after using SCI.

4.4 Genetic Local Search Algorithms

The genetic local search algorithms use two common local search algorithms, Swap and 3 Cycle respectively. We fix the parameters P_c and P_m as 0.75 and 0.25. Other components of GLS algorithms have the following options: (1) Init: ACI; (2) Cros: UX and IPPX; (3) Mut: 2D and 1D; (4) Local search algorithms: Swap and 3 Cycle.

In GLS algorithms, each offspring has to go through local search algorithms, which have a very high computation cost, so, the population size should be small. The experimental data show that (1) ACI effectively reduces the run time of local search algorithms; (2) adjacent number = 50, N = 10, and Stop crit = no improvement for 10 generations are good settings.

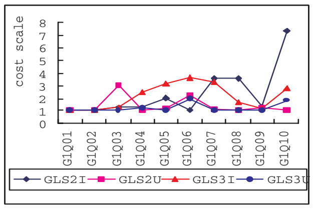
Table 4 is four GLS algorithms with Repla = keeping diversity, Init = ACI, adjacent number = 50, N = 10, and Stop crit = no improvement for 10 generations, but with different crossover operators, mutation operators and local search algorithms. Figure 5 is the costs of their optimal QEPs and Figure 6 is their optimizing times.

Table 4: GLS algorithms

GLS	Cros	Mut	Local search	Adjacent number
GLS2I	IPPX	1D	Swap	50
GLS2U	UX	2D	Swap	50
GLS3I	IPPX	1D	3 Cycle	50
GLS3U	UX	2D	3 Cycle	50

The above experimental data show:

(1) Four GLS algorithms have similar efficiency because the ratios of y-axis in Figure 5 and Figure 6 are small;



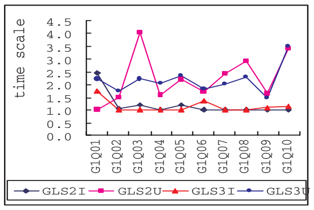
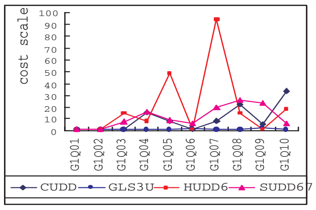


Figure 5: Costs by GLSs **Figure 6: Times of GLSs**



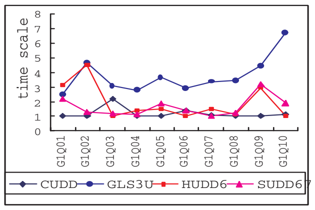


Figure 7: Costs for G1 **Figure 8: Times for G1**

(2) The ACI decreases the searching time of local search algorithm;

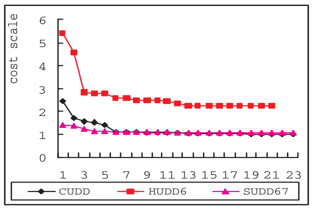
(3) GLS3U is the best GLS algorithm. The optimal settings are : N=10, adjacent number = 50, Stop crit = 10 generation of same best solution;

4.5 Comparison of Four Classes of GAs

From Section 4.1, 4.2, 4.3, 4.4, the optimal basic GA, constraint GA, heuristic GA and GLS algorithm are SUDD67, CUDD, HUDD6 and GLS3U. They all use UX as the crossover operator.

First, we compare their efficiency. Figure 7 and Figure 8 compare their costs of optimal QEPs and optimizing times. GLS3U finds the lowest cost or near lowest cost QEPs but it has the highest optimizing time. Because ratio of time is much smaller than ratio of cost, and optimizing time of GAs is smaller than execution time of QEPs, GLS3U is the most efficient GA.

Secondly, we compare their converging speeds. GLS3U is the fastest because it converges in less than 100 generations. But at each generation, each offspring has to run a local search algorithm. Therefore, we only compare the converging speeds of SUDD67, CUDD and HUDD6. For query G1Q05, Figure 9 is their optimal QEPs at 23 generations: starts from 200th generation with interval 50 gener-



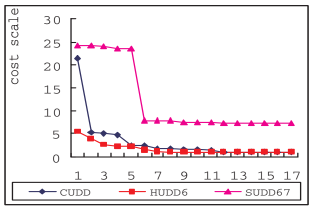


Figure 9: G1Q05 converging **Figure 10: G1Q10 converging**

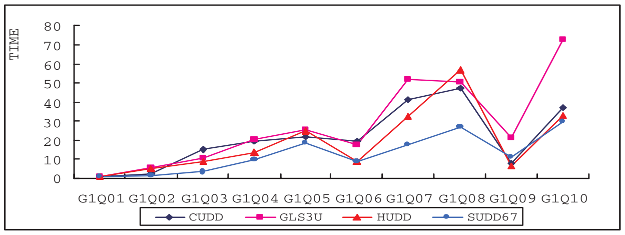
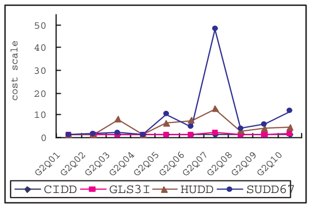


Figure 11: The time complexity of GAs



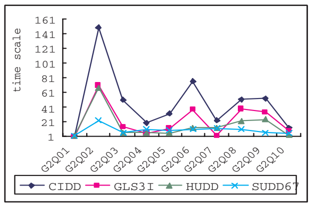


Figure 12: Costs for G2 **Figure 13: Times for G2**

ations. Figure 10 is their optimal QEPs at 17 generations: starts from 500th generation with interval 50 generations. For both G1Q05 and G1Q10, CUDD converges fast to good QEPs. SUDD67 converges fastest to a good QEP for G1Q05 but slowest to a highest cost QEP for G1Q10. So, CUDD is a stable GA for very large join queries but SUDD67 fits to queries with a smaller number of relations.

Thirdly, we compare their time complexity. Figure 11 is the time complexities. The x-axis is ten queries with relation numbers 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 respectively. The y-axis is the result of optimizing times dividing the optimizing time of same GA for G1Q01. Obviously, the time complexity of GAs increases slowly as the number of relations increases from 10 to 100.

4.6 GAs for Different Query Models

The same experimental process for G1 has been repeated for query model G2, G3, ST, SN, and MS. Figure12–Figure21 show the experimental data of four classes of optimal GAs for these five query models. Analyzing these experimental data, we find main features for five query models and explain the reasons:

1. Features of GAs for query model G2

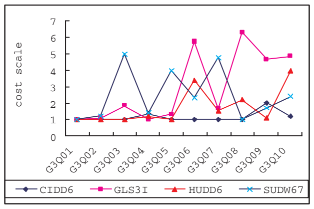
(1) Whether N=30 or N=60, the efficiency of constraint GAs and heuristic GAs are similar. It explains that ACI effectively reduces the search space;

(2) CIDD and GLS3I find the best optimal QEPs. They use ACI;

(3) SCI doesn't function, because in G2, the distribution of relation cardinalities is uniform.

2. Features of GAs for query model G3

(1) These GAs have similar optimizing times and optimal QEPs have similar costs;



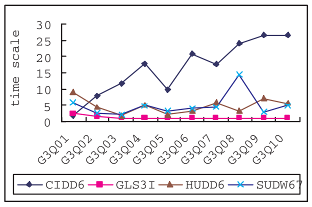
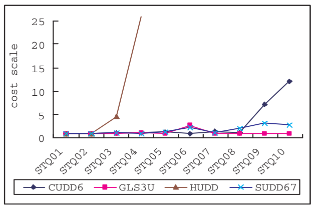


Figure 14: Costs for G3 **Figure 15: Times for G3**



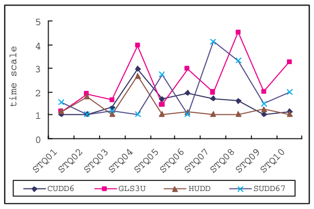


Figure 16: Costs for ST **Figure 17: Times for ST**

(2) For both constraint GAs and heuristic GAs, N=60 is better than N=30. The reason is G3 has higher connectivity than G1 and G2. There is a large number of QEPs without Cartesian-products and ACI can not reduce search space effectively;

(3) The biggest difference from G1 and G2 is that GLS algorithms prematurely converges.

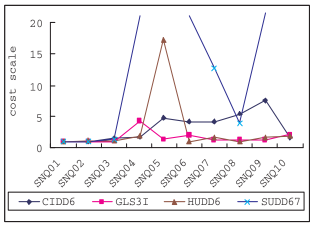
3. Features of GAs for query model ST

(1) ACI doesn't function because any non-central relation connects to the only central relation but doesn't connect to any other non-central relations. The chromosome generated by ACI includes the central relation in the first two genes. The next genes can be any remaining relations. ACI is a random initialization with the central relation as first or second gene.

(2) SCI doesn't fit, because the SCI we designed is also an ACI. Any two chromosomes in the initialized population will have the same relative order of relations except maximum three genes: their first genes, their second genes.

4. Features of GAs for query model SN

(1) ACI works better than random initialization. There is more than one central relation in query model SN, the ACI



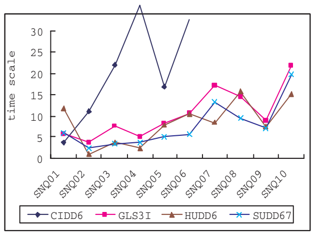
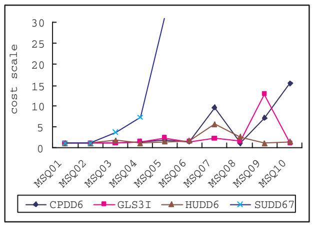


Figure 18: Costs for SN **Figure 19: Times for SN**



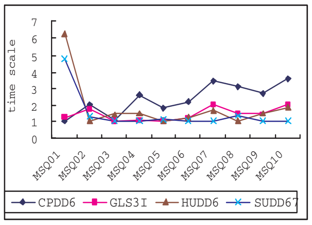


Figure 20: Costs for MS Figure 21: Times for MS

can effectively reduce a search space and keep most near optimal QEPs.

(2) SCI works better than ACI, because the former can more effectively reduce a search space;

(3) Basic GAs are not efficient because random initialization can not reduce search space;

(4) Generally, GLS algorithms are robust and efficient.

5. Features of GAs for query model MS

There are 3 to 5 central relations in query model MS. Because of the similarity between SN and MS – more than one central relation – the ACI and SCI function better than random initialization in query model MS. A important feature of GAs for query model MS is that HUDD6 and HUDD are very robust.

Table 5 is the rank of efficiency of four classes of GAs for six query models. Obviously, the rank of efficiency of four classes of GAs is different for six query models and the optimal settings of four classes of GAs are different for different query models.

Table 5: Efficiency of GAs for six query models

	BGA (Rank)	**CGA (Rank)**	**HGA (Rank)**	**GLS (Rank)**
G1	SUDD67(3)	CUDD(2)	HUDD6(4)	GLS3U(1)
G2	SUDD67(4)	CIDD(2)	HUDD(3)	GLS3U(1)
G3	SUDW67(3)	CIDD6(1)	HUDD6(2)	GLS3I(4)
ST	SUDD67(2)	CUDD6(3)		GLS3U(1)
SN	SUDD67(4)	CIDD6(3)	HUDD6(2)	GLS3I/U(1)
MS	SUDD67(4)	C*DD6(3)	HUDD6(1)	GLS3I/U(2)

5. CONCLUSIONS

Because genetic algorithms are randomized optimization algorithms, their efficiency depends on the nature of the problem. We believe that all large join queries have a too wide a range of complexity and diversity to be based on to optimize GAs for LJQO. Our experiments show that a query model is a proper granularity to be based on to optimize GAs for LJQO. Based on three typical query models of relational database systems and three typical query models of data warehouse systems, we choose a left-deep tree space as GAs' search spaces to design and optimize four classes of GAs for LJQO. Our experimental results have proved this from the following two aspects. First, the efficiency of four classes of GAs depends on query models. Second, the efficient settings of GAs depend on query models.

6. ACKNOWLEDGMENTS

We acknowledge the assistance of the computer science department, Memorial University, St. John's, Newfoundland, Canada, in the writing and preparation of this paper. This research is funded in part by the National Natural Science Foundation of China – NFSC60573038.

7. REFERENCES

[1] Bӑck, T., Fogel, D. B., Whitley, D. and Angeline, P. J. Mutation operators. Evolutionary Computation 1 Basic Algorithms and Operators. Institute of Physics Publishing, Bristol and Philadelphia, 2000, 237-254.

[2] Bennett, K., Ferris, M. C., and Ioannidis, Y. A genetic algorithm for database query optimization. Tech. Rep. TR1004, Univ. Wisconsin, Madison, 1991

[3] Bierwirth, C., Mattfeld, D. C., and Kopfer, H. On Permutation Representations for Scheduling Problems. In Parallel Problem Solving from Nature , H.-M.Voight et al, eds. Springer-Verlag, 1996, 310-328.

[4] Booker, L. B., Fogel, D. B., Whitley, D. and Angeline, P. J. Recombination. Evolutionary Computation 1 Basic Algorithms and Operators. Institute of Physics Publishing, Bristol and Philadelphia, 2000, 256-307.

[5] Eshelman, L. J. Genetic algorithms. Evolutionary Computation 1 Basic Algorithms and Operator. Institute of Physics Publishing, Bristol and Philadelphia, 2000, 64-80.

[6] Gorges-Schleuter, M. On the power of evolutionary optimization in the examples of ATSP and large TSP Problems. European Conference on Artificial Life. Brighton, UK, July 1997.

[7] Ioannidis, Y. E. and Kang, Y. C. Randomized Algorithms for Optimizing Large Join Queries. In Proceeding of the 1990 ACM-SIGMOD Conference on the Management of Data. Atlantic City, NJ, .May 1990. 312-321.

[8] Lahiri, T. Genetic Optimization Techniques for Large Join Queries. In Proceedings of the Third Genetic Programming Conference (Univ. of Wisconsin, Madison, July 22-25, 1998). Morgan Kaufmann, 1998, 535-540.

[9] Schwefel, H. Advantages (and disadvantages) of evolutionary computation over other approaches. Evolutionary Computation 1 Basic Algorithms and Operators. Institute of Physics Publishing, Bristol and Philadelphia, 2000, 20-22

[10] Steinbrunn, M., Moerkotte, G., and Kemper, A. Heuristic and randomized optimization for the join ordering problem. The VLDB Journal,6, 3 (Aug. 1997), Springer-Verlag, New York, Inc. 191-208

[11] Stillger, M. and Spiliopoulou, M. Genetic programming in database query optimization. In Proc First Annu. Conf. Genetic Programming, Stanford, CA, July 1996, 388-393

[12] Swami, A. and Gupta, A. Optimization of Large Join Queries. In Proceedings of the 1988 ACM SIGMOD International Conference on the Management of Data. SIGMOD RECORD Volume 17. Number 3, September 1988, 8-1