Parallel Query Optimization: Exploiting Bushy and Pipeline Parallelism with Genetic Programs

Michael Stillger[a] Myra Spiliopoulou[b] Johann-Christoph Freytag[a]

[a] Institut für Informatik
Humboldt-Universität zu Berlin
{stillger,freytag}@dbis.informatik.hu-berlin.de

[b] Institut für Wirtschaftsinformatik
Humboldt-Universität zu Berlin
myra@wiwi.hu-berlin.de

Abstract

Parallel query optimization is one of the hardest problems in the databases area. The various cost models reflecting the query execution parameters determine the structure and size of the solutions space. To explore this space, research has turned towards combinatorial optimization techniques, heuristics and genetic algorithms, which have been primarily studied for sequential query processing.

In this study, we propose a genetic programming strategy for the optimization of parallel bushy query execution plans. Genetic programming has evolved from genetic algorithms, and is more flexible and expressive. We consider two cost functions modelling different modes of interoperator parallelism. We analyse the behaviour of the search strategy and observe that it is affected by the cost function. Despite this, our experiments show that our strategy converges to optimal plans of very good quality, and performs best when bushy interoperator parallelism is exploited.

Keywords. Genetic programming, optimization strategies, parallel solution spaces, query cost modelling, bushy parallelism, pipeline.

1 Introduction

The key to success for a database system is the effectiveness of its optimizer in producing efficient query execution plans (QEPs). Database parallelism has the potential of increasing the execution efficiency, but it makes the query optimization problem more complex. In this study, we present an optimization technique based on the genetic programming paradigm, which suits better the optimization problem domain than a genetic algorithm. Our technique exploits the advantages of evolutional computation [Hol75] while avoiding the encoding complexity and the pitfalls of genetic algorithms for bushy query execution plans. We have studied the behaviour of our technique for different parallel solutions' spaces and found it to be fully competent to a classic search strategy.

Since all but the most trivial database queries involve joins, there is a lot of research focussing on the optimization of join queries. Large join queries, occuring in decision support systems, knowledge bases and advanced database applications, including CAD/CAM, are of particular interest, and many researchers have proposed techniques for their efficient optimization [SG88, IK91, LVZ93, LOY94, SHC96].

The solutions space for parallel join queries is too large to be scanned exhaustively. Techniques based on combinatorial optimization are recently being tested on parallel architectures [LVZ93, LOY94, SHC96], after having been studied extensively for sequential queries [SG88, IK91, SMK93]. Despite their promising results, their effectiveness is affected by the shape of the solutions space: Iterative improvement might be inefficient in a space containing high plateaus and local minima at various elevations [IK91, LVZ93]. Simulated annealing is subject to the same disadvantages but with a lower probability. On the other hand, simulated annealing is substantially slower than iterative improvement [SG88, IK90].

The robustness of genetic algorithms towards problems with solutions spaces of unknown shape has motivated the reasearchers to adapt them for the query optimization problem [BFI91, L$^+$91, SMK93]. Genetic algorithms are applied on chromosomes. Therefore, they require the transformation of the problem "states", the QEPs for query optimization, into string structures. In [BFI91, L$^+$91, SMK93], methods are presented to transform QEPs into chromosomes, on which crossover, selection and mutation are applied. To compute the fitness, though, a chromosome must be transformed back to a QEP on which the cost

function is applied. Those transformations contribute considerably to the high optimization overhead. Moreover, their crossover operator, adapted from [Gol89], does not exactly obey the semantics of the building blocks for bushy QEPs and may destroy the tree structure of a chromosome. So, the new QEP has lost a lot of the structural information of its ancestor.

Despite these shortcomings, the behaviour of genetic algorithms is satisfactory [BFI91, SMK93]. This indicates that evolutional computation, of which genetic algorithms are a special instance, is appropriate for query optimization. In this study, we propose a genetic *programming* technique; we show that it is closer to the nature of the problem, avoids the complexity and ambiguity of genetic algorithms' techniques and converges into optima of good quality.

We developed our *G*enetic *P*rogramming (GP) technique for large join queries in parallel spaces. We consider interoperator parallelism in its full potential by considering bushy QEPs. We model the parallel optimization problem (a) when only bushy parallelism is supported, and (b) when both bushy and pipeline parallelism are taken into account.

In the next section, we briefly introduce the basic principles of genetic programming. In section 3, we present our optimization strategy, adapting the genetic programming paradigm to the parallel query optimization problem. In section 4, we present two cost functions modelling different aspects of interoperator parallelism, and we study the cost distribution in their solutions spaces. Then, in section 5, we analyze the behaviour of our technique for those cost functions, and we compare it to a reference technique based on combinatorial optimization. Our results show that our GP-technique is competent in terms of effectiveness and efficiency. The last section concludes our study.

2 Genetic Algorithms and Genetic Programming

2.1 Genetic Algorithms

The Genetic Algorithm (GA) paradigm [Hol75] applies strategies adapted from the natural process of evolution to a complex search problem. A possible solution of the search problem is encoded as a string of fixed length, a "chromosome" representing an "individual" in the GA. A set of individuals forms a "population". The population at a particular point in time is a "generation".

The GA paradigm is based on four operators applied on the individuals of a generation, in order to produce the next generation. The first operator computes the "fitness" value of each individual. This value is based on the cost function of the particular problem. It represents the relative quality of the individual within its generation.

The second operator is the "selection", which materializes the concept of "survival of the fittest". Based on the relative fitness of the individuals, it selects parents from the population, on which the crossover and mutation operators described below are applied to produce the next generation.

The "crossover" is a binary recombination operator. It randomly selects a substring from each of the two parent individuals, it exchanges the two substrings and produces two offspring individuals that have inherited parts from both parents. The selection of the substrings must occur in such a way, that the *building blocks* of the individual are not destroyed. The building blocks represent properties that determine the individual's fitness. If a selected pattern in the encoded string does not match a property of the decoded individual, then the connection between representation and fitness is lost. The absence of inherited tree similarities of bushy QEPs observed in [BFI91] was due to the crossover operator selecting substrings that did not necessarily correspond to correct subtrees.

The "mutation" is a unary operator that applies a small change on an individual. The new individual thus produced brings new genetic material to its generation.

Given the four operators, the complete run of a genetic algorithm can be outlined as follows:

1. Create a random population

2. Evaluate the fitness of each individual in the population

3. Create the next generation by applying crossover and mutation to the individuals of the population, until the new population is complete. A selected parent is not removed from the population, so it can be selected multiple times.

4. Stop when a termination criterion (time, number of generations) is met. Otherwise, go to step 2.

Several alternative methods for the GA operators have been proposed in the literature, as well as more sophisticated recombination methods. An overview can be found in [Gol89, Mic94].

2.2 Genetic Programming

The main differences between GA and Genetic Programming (GP) [Koz91] lay in the representation of the individuals and, consequently, in the operators on them: GA demands a string representation, while GP uses a tree representation. The latter is often closer to the structures of the specific problem domain and allows the design of more efficient operators.

In GP, a tree stands for an abstract computer program, similarly to program representation in compilers: The program tree consists of a set of functions (nodes) and a set of terminals (leaves). The fitness is a function of the output value of this program. Koza uses LISP parse trees where the name of a LISP function is a node and its arguments are either the child subtrees of the next level or leaves (terminals) [Koz91]. The crossover exchanges randomly selected *subtrees* of the parent individuals. In contrast to the fixed length chromosomes, trees can grow and shrink. More important, each subtree being incorporated into a new child individual still maintains its semantics as a subprogram.

A parallel query execution plan can be observed as a programm for the database execution engine. The tree structure of QEPs is thus by nature close to the abstract computer program represented as a tree in GP. In the next section, we describe how we adapt GP to the modelling of QEPs and design a search strategy for the solutions space of parallel QEPs.

3 A Search Strategy based on Genetic Programming

We introduce a search strategy for parallel solutions spaces, which is based on the genetic programming paradigm. Differently from the genetic algorithms approach, genetic programming does not require the unintuitive transformation of QEPs into chromosomes. Rather, it is possible to exploit the advantages of the evolution paradigm, while maintaining the original QEP structure.

3.1 The QEP Representation

Join trees. A Query Evaluation Plan (QEP) is a tree representation of a declarative query, augmented with execution directives. The leaves of the tree are database relations, the non-leaf nodes are operators, and the edges represent the data streams flowing towards the nodes. Since our study focusses on join queries, the non-leaf nodes of our QEPs are always join operators.

It has been shown that the optimal QEP for sequential queries is most often located in the solutions space of bushy QEPs [BFI91, IK91, SMK93]. Given the importance of bushy QEPs for the exploitation of bushy parallelism, we expect this observation to hold in parallel spaces as well. Therefore, we consider bushy QEPs in our model. We consider both forms of interoperator parallelism [Gra93]: Sibling nodes/subtrees can be executed simultaneously, thus supporting bushy parallelism. Edges between consecutive nodes can be blocking or pipelining [HM94], so that pipelining can be exploited.

In Figure 1, we show an example query graph, and in Figure 2, we present two QEPs for this graph. Non-leaf nodes are surrounded by a circle. All nodes and leaves are identified by their join, respectively relation, number. The join-nodes are labelled with the execution algorithm; we consider the three classic join algorithms, nested-loops 'n', hash-join 'h' and merge-sort 'm'. In order to keep the figures simple, we omit all other execution directives, such as access methods, as well as the leaf relations. We only show the labels and the leaves for the marked subtrees S_1 and S_2, which we use in subsequent examples.

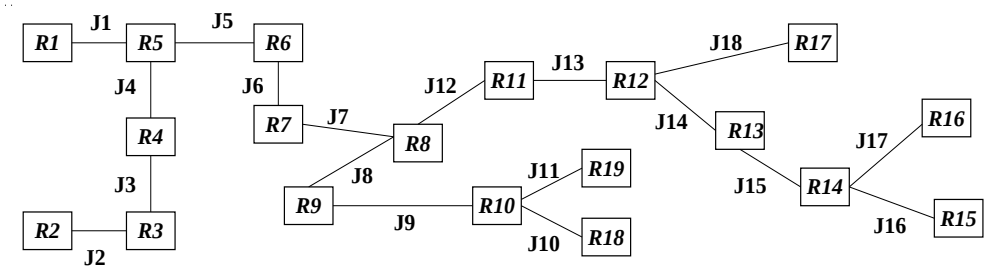

Figure 1: Query Graph for a 18 Join Query

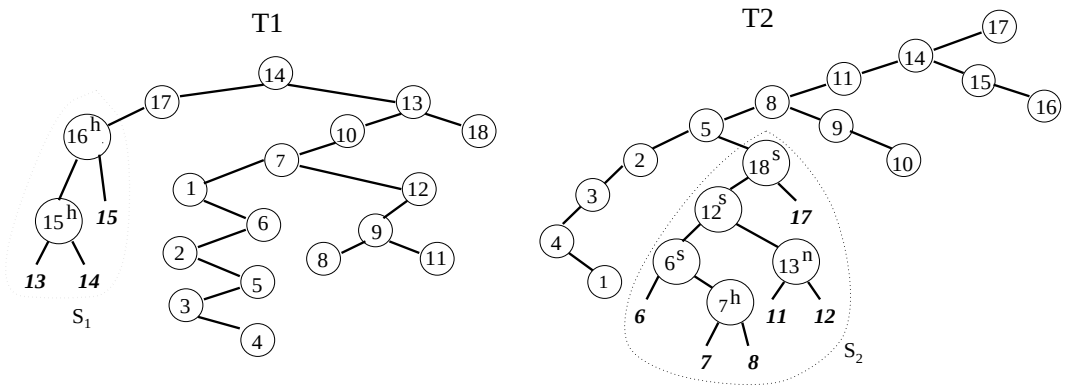

Figure 2: Join trees T_1 and T_2

Valid trees. A solutions space consisting of all possible QEPs also contains trees with cartesian products and "forced intersections". "Forced intersections" occur when the same relation participates in joins placed in different subtrees: the output streams of these joins must be intersected to produce the tuples satisfying the conjunction of the join predicates.

Cartesian products and forced intersections are expensive operations. Although they may increase the degree of bushy parallelism, they also increase the size of the solutions space and the complexity of the QEP transformations. We therefore introduce the notion of *valid* QEP and *valid* tree:

Definition 1: A QEP is *valid* iff its join tree is valid. A join tree is *valid* if:

1. It contains no cartesian products: the left, respectively right, relation of each join node is referenced in the left, respectively right, subtree below that node.

2. It contains no forced intersections: each relation and each join-node appear only once in the tree.

Obviously, if a tree is valid, all its subtrees are valid. In the following, we use the terms (valid) "QEP" and "tree" interchangeably. The two QEPs T_1 and T_2 in Fig. 2 are valid.

Hereafter, we consider only valid QEPs. Since query optimizers hardly ever consider cartesian products (for an exception see [CM95]), and since forced intersections imply additional processing overhead, restricting the search space to that of valid QEPs does not limit the applicability range of our approach.

A valid QEP as an abstract computer program. A QEP can be observed as a genetic program in an abstract tree representation, which is evaluated in a bottom-up way by the database system. The relations are the terminals and the joins are the functions of the genetic program. Due to the closure property of the relational model, the input and the output of all operators in a QEP are relations. Thus, the closure property defined in [Koz91] (chapter 6.1.1) is also satisfied: each of the functions in the function set must be able to accept as argument any value and data type structure returned by any other function or assumed by any terminal in the terminal set. A QEP satisfies the structural requirements of the genetic programming method.

The random placement of nodes and subtrees performed by the classical GP crossover and mutation can introduce cartesian products and forced intersections. Since we prohibit them in the space of valid QEPs, we design our crossover and mutation in such a way that they always produce valid QEPs, while still allowing the random placement of subtrees.

3.2 Crossover and Mutation

Crossover. The crossover of the GP model is applied on two trees from which two subtrees are selected for exchange. It produces two trees of the next generation, which must satisfy the following requirements:

1. They are valid according to Definition 1.
2. Each of them (a) inherits (most of) the structure of one parent and (b) contains the selected subtree of the other parent.

We introduce the α-operator for tree $\underline{\alpha}$ssembling [1] and use it to define our operators. All trees referred to in the following definitions describe the same query issued to the system.

Definition 2: $\alpha(nodelist, T\_set)$: Let *nodelist* be an ordered list of join nodes appearing in a QEP's tree, and let T_set be a set of subtrees. The α-operator returns a new content of T_set, comprised of subtrees assembled together in the following way [2];

```
foreach join J in nodelist do
    If there is a subtree S ∈ T_set with J appearing among its nodes
        then skip J and continue the loop;
    Find a subtree S1 ∈ T_set that references the left relation of J
    Find a subtree S2 ∈ T_set that references the right relation of J
    If no S1 and no S2 are found
        then set T_set = T_set ∪ {J} and continue the loop;
    αssemble a new tree S,
        where J is the root, S1 (if any) is the left child and S2 (if any) is the right child;
    Remove S1, S2 (if any) from T_set;
    Insert S into T_set;
end-foreach;
```

According to this definition, the join nodes in the *nodelist* are iteratively selected and attached to the subtrees already in T_set, if possible. Otherwise, they are added to the T_set as dangling nodes to be brought together later. When the operation completes, T_set consists of several subtrees corresponding to parts of the original query. We show that for specific initial contents of *nodelist* and T_set, the α operator produces a final T_set consisting of a single valid query tree.

We first define the notion of *postorder_joinlist* and *leaves_of*:

Definition 3: *postorder_joinlist(T)* is the list of join nodes produced by traversing (sub)tree T in postorder, i.e. in leftChild–rightChild–root order.

Definition 4: *leaves_of(T)* is the set of relations appearing in the (sub)tree T.

Definition 5: $crossover(T_1, S_1, T_2, S_2)$ is applied on two valid trees T_1 and T_2, from which two subtrees S_1, respectively S_2, are selected. It produces two valid trees NG_1 and NG_2 of the next generation by invoking the α-operator:

$$NG_1 := \alpha\left(postorder\_joinlist(T_1), \{S_2\}\right)$$

[1] A similar operator is proposed in [BFI91] for the transformations between a tree and its chromosome representation.

[2] For notational simplicity, we distinguish between the initial T_set used as *argument* to the operator, and the final T_set observed as *return value* of the operator. In the implementation, the T_set argument is simply passed by reference.

$$NG_2 := \alpha\,(postorder\_joinlist(T_2), \{S_1\})$$

In Appendix A we prove that the $\alpha(\cdot)$ operator, as invoked for crossover, outputs a T_set consisting of a single valid tree for the initial query.

Example 1: In the trees of Fig. 2, we have marked the subtrees S_1 and S_2 selected for the crossover of T_1 and T_2. The postorder joinlist of T_1 is (J15, J16, J17, J4, J3, J5, J2, J6, J1, J8, J11, J9, J12, J7, J10, J18, J13, J14). Of these, the join-nodes already belonging to S_2 are ignored by the α operator, according to the first test in the loop described in Def. 2. According to Def. 5, the new tree NG_1 is produced as:

$$NG_1 = \alpha\,((J15, J16, J17, J4, J3, J5, J2, J6, J1, J8, J11, J9, J12, J7, J10, J18, J13, J14), \{S_2\})$$

By selecting the join nodes from the postorder joinlist and connecting them to form NG_1, we restore T_1 as far as possible: The nodes subtracted from the postorder list are those belonging to S_2. However, S_2 is an element of T_set; as such, it is built into to NG_1 intact. Thus, NG_1 inherits most of the structure of T_1 and contains S_2 in its original form.

In Fig. 3, we show the first step of the α-operator that finally produces NG_1. Join $J15$ is the first one to be processed. Since it does not reference the same relations as S_2, it becomes a separate element of T_set. In the next step, it will become the left child of join $J16$, whose left input relation is $R14$. NG_2 is created in a similar way. Fig. 4 shows the trees NG_1 and NG_2 produced by crossover.

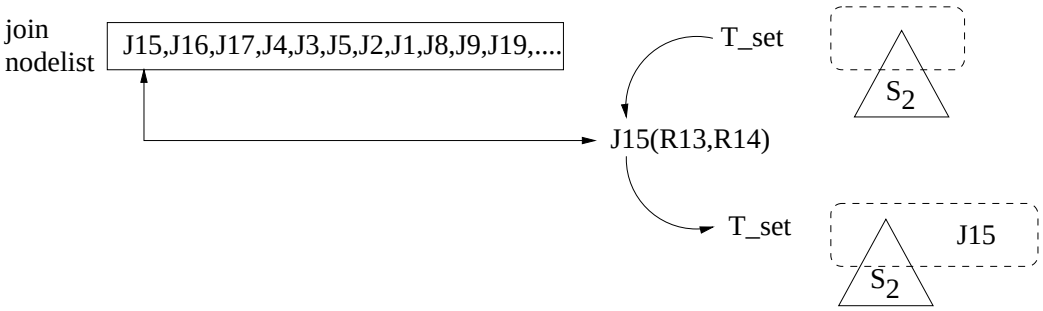

Figure 3: Applying crossover on tree T_1

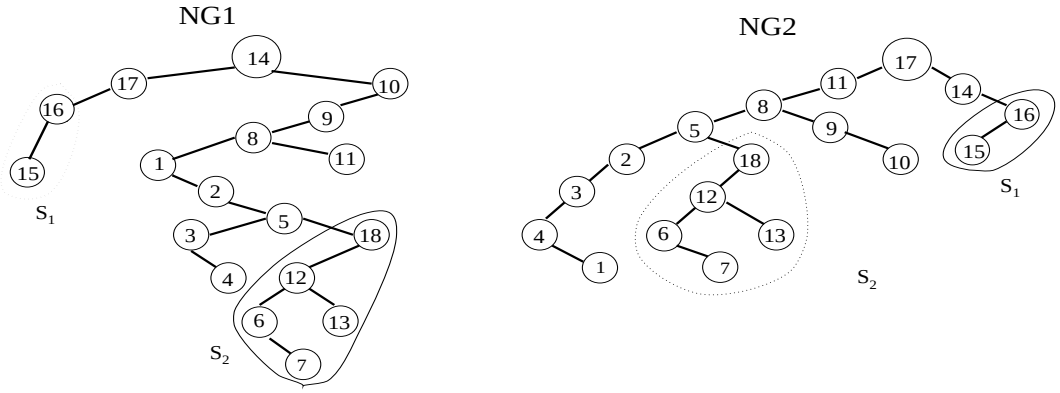

Figure 4: Next Generation: NG_1 and NG_2

The trees produced by crossover inherit thus most of the structural characteristics of their parents. The subtrees selected for a crossover appear intact in the new trees. Moreover, if the selected subtrees S_1, S_2 have no nodes in common, crossover rebuilds S_1 in NG_1 exactly as it was in T_1 (same for S_2 in NG_2). Hence, *both* subtrees appear intact in *both* trees of the next generation.

Our crossover ensures in an elegant way that a QEP is combined with the structure and with the original join nodes and algorithms of another QEP into a valid offspring. Hence, the requirement of the evolutional process, that building blocks are kept and recombined, is satisfied.

It should be stressed that the crossover does not simply change the tree structure of a QEP. The subtree transferred from one QEP to another carries with it the algorithm assignments and the settings of the edges as blocking or pipelining (terminology in [HM94]). In that sense, crossover has a more complex impact on a QEP than the QEP transformations proposed in [IK91, SHC96] for combinatorial optimization techniques.

Mutation. We consider two mutation operators:

- $mutate_1(T, newAlg)$ changes the join algorithm into $newAlg$ for a randomly selected node in tree T. The structure of T is not affected, but its cost and fitness do change.
- $mutate_2(T)$ swaps the position of two randomly selected nodes of a tree T and reconstructs a valid tree NG of the next generation:

$$mutate2(T) := \alpha(swap(postorder\_joinlist(T)), \emptyset)$$

3.3 The Fitness Function and the Cost Function

The fitness function of our GP model is based on the cost function of the query optimizer. The cost of each individual QEP in the population is calculated, and then its fitness is computed as the normalized execution cost: the least expensive QEP is the fittest one and has fitness 1.

For the next generation, individuals are selected according to their fitness. The fitter individuals are more likely to produce children of the next generation. In addition, building blocks of good quality can be recombined from generation to generation. Thus, the algorithm converges towards generations with low cost individuals. The optimal QEP is then the best individual among all generations.

4 Cost Models and their Solutions Spaces

The search strategy was described thus far independently of the parallel architecture and the types of parallelism being exploited, although the GP operators do affect the exploitation of parallelism on the QEP. The characteristics of parallelism being exploited are expressed in the cost model. In this work, we study the behaviour of our GP strategy for parallel search spaces, (a) when only bushy parallelism is exploited and (b) when pipeline is also considered. Accordingly, we have developed two cost models, one for bushy parallelism and one for bushy and pipelined parallelism, and studied their impact on the behaviour of our search strategy.

In these models, the "cost" of a QEP is the total elapsed time from the beginning to the completion of query execution. Therefore, the cost of two processes running simultaneously is the cost of the slowest one. Two different generalizations of those cost models are presented in [SHC96, SF96].

4.1 Assumptions

We consider a shared-nothing or shared-disk parallel machine or LAN. The processors are homogeneous, with main memories of equal size. The execution time is measured as I/O and data transfer time. Given the speed of modern processors, we can assume that CPU cost is negligible. We only consider the qualitative impact of multiprocessing and assume that there are enough processors to avoid multitasking. Hence, we do not need to incorporate scheduling information in the cost models.

In a shared-nothing system, data locality is a complicating factor. To keep the cost models simple, we have assumed that non-shared disks are used to store intermediate relations only, while the base relations are located in a data pool serviced by a disk or disk array controller that is fast enough to prevent a data retrieving process from impedding the data retrieval of another.

The data transfer unit for disk and network is a "page" of system-defined size. We denote by t_{disk} the time required to access a page in a local disk, and by t_{net} the time required to transmit a page across the network.

4.2 Join cost

We focus on interoperator parallelism, so that each join x is executed by a single processor. The execution time $T(x)$ consists of the time $T_{in}(x)$ required by the processor to retrieve the input streams, the disk access cost $T_{local}(x)$ occuring during local processing of data not fitting in main memory and the time $T_{out}(x)$ required to store or forward the output stream.

$$T(x) = T_{in}(x) + T_{local}(x) + T_{out}(x) \tag{1}$$

The output stream of x is either stored locally or transmitted to another processor. Let $sf(x)$ be the selectivity factor of x, L be the size of the left input stream in pages and R the size of the right input stream. The output cost is:

$$T_{out}(x) = sf(x) \cdot L \cdot R \cdot \begin{cases} t_{disk} & \text{, if the output is written to disk} \\ t_{net} & \text{, if the output is sent to another processor} \end{cases} \tag{2}$$

The streams input to x are retrieved from a local or a remote disk or from the network, depending on whether data are stored-and-forwarded or pipelined. The two streams can be read in parallel at cost:

$$T_{in}(x) = \max(retrieve(L), retrieve(R)) \tag{3}$$

where the function $retrieve()$ for a number of pages p is defined as:

$$retrieve(p) = \begin{cases} p \cdot t_{disk} & \text{, retrieval from the local disk} \\ p \cdot t_{net} & \text{, retrieval across the network} \\ p \cdot (t_{net} + t_{disk}) & \text{, retrieval from a remote disk} \end{cases}$$

For the execution of a join, we consider the nested-loops, merge-sort and hash-join algorithms. For $T_{local}(\ldots)$, we use the formulae presented in [SHC96].

4.3 A Cost Model for Bushy Parallelism

Our first cost model, hereafter denoted as "BO", exploits *b*ushy parallelism *o*nly. Joins appearing in different subtrees of the QEP are executed independently, as soon as their input streams are available for processing. Thus, the cost of the QEP is the cost of the most expensive branch.

In this execution scenario, the output of a join must be stored in the local disk, from which it is retrieved by its consumer. Hence, the cost of a join consists of the time needed to retrieve its input streams from a local or remote disk, the time for local processing, and the time for storing the output stream locally:

$$T^{BO}(x) = T_{in}(x) + T_{local}(x) + T_{out}(x)$$

according to Eq. 1, where $T_{in}(x)$ and $T_{out}(x)$ consist of disk access cost for remote and for local accesses, as defined in Eq. 3 and 2 respectively.

The execution cost of the tree rooted at x is the total elapsed time $C^{BO}(x)$ from the beginning of query execution until the completion of join x [3]. The cost is equal to the execution time of x and the time required by its slowest producer to complete execution:

$$C^{BO}(x) = T^{BO}(x) + \begin{cases} 0 & \text{, } x \text{ is a leaf} \\ C^{BO}(y) & \text{, } x \text{ has one child, join } y \\ \max\left(C^{BO}(y), C^{BO}(z)\right) & \text{, both children of } x \text{ are joins } y, z \end{cases}$$

Hence, the execution time of a QEP is the time required for its root process r to complete execution, $C^{BO}(r)$.

[3] In this section, we use the conventional notion of a "leaf", as a join whose input streams are base relations. A non-leaf join has at least one child/producer that is another join.

4.4 A Cost Model for Bushy Parallelism and Pipelining

Our second cost model, hereafter denoted as "B&P", exploits *b*ushy parallelism and *p*ipelines. Pipelining is modelled on a QEP by labelling the edge between the child/producer and parent/consumer node as "blocking" or "pipelining". According to Hasan and Motwani [HM94], an edge between a consumer and a producer is blocking, if the consumer must wait until the producer has generated the whole output stream. If the output stream can be processed tuple-by-tuple, then the edge is pipelining. We use the same definition, except that the data transfer unit is the page and not the tuple. Similarly to [HM94], we consider pipelines without delays. If a pipelining edge introduces delays [WA91], we assume that the optimizer turns it into a blocking edge.

The two streams input to join x are read in parallel and their retrievals are overlapped. Since intermediate data streams are not materialized to disk, the cost of sending a stream is overlapped by the cost of receiving it. Hence, the cost of a join x is the time needed to read and process its input streams:

$$T^{B\&P}(x) = T_{in}(x) + T_{local}(x)$$

Similarly to [HM94], we assume that pipelines are perfect, i.e. there are no processing delays the processes in a pipe. Hence, the cost of a branch composed of pipeline edges is the cost of the most expensive join in the pipe [HM94]. The cost of a node whose output is received across a blocking edge is the time needed to compute the whole output stream. This time is added to the cost of the branch. As for the BO model, the cost of the QEP is then the cost of the most expensive branch.

Hence, the elapsed time for the execution of the tree rooted at x has two factors: pipeline cost $PC(x)$ and blocking cost $BC(x)$. Intuitively, if x has a child connected to it with a blocking edge, $BC(x)$ is the time needed by this subtree to complete execution. If x has a child connected to it with a pipeline edge, then x runs in parallel with its child; $PC(x)$ is the execution time of the slowest node in the pipe. More formally, $PC(x)$ and $BC(x)$ are computed recursively as follows:

- The streams input to x are base relations. Then all its cost is pipeline cost:

$$\begin{aligned} PC(x) &= T^{B\&P}(x) \\ BC(x) &= 0 \end{aligned}$$

- Node x has one or two children, y_1 and y_2. The pipeline cost is:

$$PC(x) = \max_{i=1,2} \left(T^{B\&P}(x), pc_i \right) \tag{4}$$

where

$$pc_i = \begin{cases} PC(y_i) & , (y_i, x) \text{ is a pipeline edge} \\ 0 & , \text{otherwise} \end{cases}$$

The blocking cost of x is:

$$BC(x) = \max_{i=1,2} \left(BC(y_i) + bc_i \right) \tag{5}$$

where

$$bc_i = \begin{cases} PC(y_i) + T_{out}(y_i) & , (y_i, x) \text{ is a blocking edge} \\ 0 & , \text{otherwise} \end{cases}$$

Example 2: In Fig. 5, we show a (sub)tree rooted at node x. x is connected with a pipeline edge to its left child yL; we mark this edge with an arrow. The edge connecting x to R is blocking. The triangles represent the subtrees below yL, yR. This example could represent a hash-join process x, whose building relation comes from the right child yR.

Since the edge (x, yR) is blocking, x cannot start before yR, and thus the whole subtree sR, completes. The execution cost across this edge is:

$$rBlocking = BC(yR) + PC(yR) + T_{out}(yR)$$

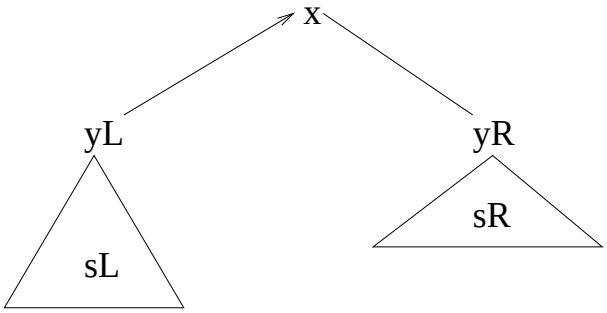

Figure 5: Estimating the cost of an example tree

where $BC(yR)$ incorporates all blocking cost of sR, and $PC(yR)$ incorporates the pipeline cost of sR and the processing cost of yR.

Since the edge (x, yL) is pipeline, x runs in parallel with yL. The pipeline cost of the (sub)tree rooted at x is the cost of the most expensive pipe member:

$$PC(x) = max(T^{B\&P}(x), PC(yL))$$

where $PC(yL)$ is itself the maximum between $T^{B\&P}(yL)$ and the pipeline cost below it.

The blocking cost contributed by yL is only the blocking cost of the underlying subtree sL, if there are any blocking edges there: $lBlocking = BC(yL)$, where $BC(yL)$ incorporates the cost of blocking edges within sL.

The blocking cost of x is the time x has to wait for nodes below blocking edges to be processed. This time is the maximum waiting time between the left and right child of x:

$$BC(x) = \max(lBlocking, rBlocking)$$

The cost of executing the QEP is the sum of the blocking cost and the pipeline cost of the entire tree, and of the output cost of the root process r:

$$C^{B\&P}(r) = PC(r) + BC(r) + T_{out}(r)$$

The factor $T_{out}(r)$ is added, because r has no consumer to read its output and overlap the output cost.

A main-memory variation. In the B&P cost model, disc accesses occur whenever the edge between two join nodes is blocking and when the intermediate results of a join operation do not fit in main memory. For large data streams, this overhead can become a determinant factor. In order to extrapolate the impact of pipelining edges on query execution cost, we have implemented a variation of the B&P model, in which the main memory is large enough to hold any intermediate results. Hence, the noise caused by the disc access overhead is eliminated, and the determinant factor of QEP cost is the cost of processing data in parallel and pipelined mode.

This cost model variation, hereafter denoted as B&P-X, has been used as a third cost model in the subsequent experiments.

4.5 The Solutions Spaces of the Cost Models

Before studying the behaviour of our genetic programming strategy for the different cost models, we have studied the cost distribution in their solutions spaces. Ioannidis and Kang showed that the cost distribution of acyclic queries in the solutions space of bushy sequential QEPs follows the γ-distribution [IK91]. Our experiments indicate that the parallel solutions spaces are rather different.

Parameters of the experiments. We have studied the cost distribution for 10 query sizes. For each size, we have generated an acyclic query graph. The database parameters are summarized in Table 1. We consider two databases, the Small and the Large one, containing relations with different size ranges. Our database and query settings are close to those used in "portofolio" database experiments, as presented in [LVZ93].

The settings of the parallel architecture assumed by the cost models are shown in Table 2. The small size of processor memory was intended to counterbalance the modest size of the database relations, in the sense that main memory should not be adequate to hold all relations.

Relation sizes:	
	Small database: 1,000 to 10,000 tuples
	Large database: 1,000 to 100,000 tuples
Attribute sizes:	8 - 20 bytes
Output attributes:	4
Number of joins:	10 - 100

Table 1: Database and Query parameters

Page size:	1024 bytes
Page transfer time - Network:	1.7 msec (600 Kbytes/sec)
Page transfer time - Local Disk:	8.3 msec
Page transfer time - Remote Disk:	10 msec
Number of processors:	100
Processor memory:	800 Kbytes

Table 2: Parallel machine parameters

Cost distributions. We have computed the cost distribution for each cost model, using a sample of 50,000 to 75,000 QEPs per query depending on the query size. In Figures 6, 8 and 10, we show our results for the Small database. Figures 7, 9 and 11 contain our results for the Large database. The horizontal axis contains QEP cost values, and the vertical axis the number of QEPs corresponding to each cost value.

The Figures show that the database has a remarkable impact on the cost distribution. For the Large database, the cost distribution reveals high concentrations of QEPs and low diversity of values. This is caused by the disk I/O, which turns to be the dominant factor of database cost.

The base relations input to the leaf nodes, all intermediate results not fitting in main memory, and the output streams for the BO model, imply I/O accesses. The I/O cost for processing a very large (intermediate) relation can thus easily become the dominant factor, especially for small queries. Therefore, QEPs processing the same relation can have the same cost, although they may differ in the rest of their structure. For the Small database, the diversity is higher and a wide spectrum of values is covered smoothly. This is due to the lower diversity of relation sizes occuring in this database.

concentrations' effect for the B&P cost model. For the B&P-X cost

5 Behaviour of the Genetic Programming Strategy

5.1 A reference technique

To study the performance of our strategy, we need to compare it with a reference strategy. Exhaustive search in a parallel solutions space is not possible for the given range of query sizes, so that a non-exhaustive reference technique must be used. In [GLPK94], it has been claimed that simple random generation of QEPs shows faster convergence than a combinatorial optimization technique, like iterative

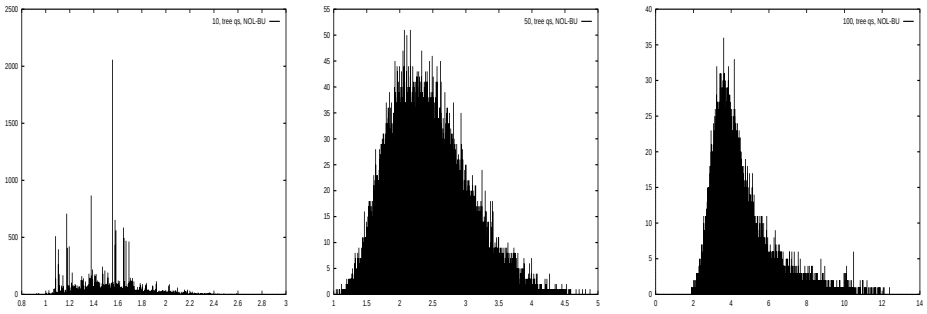

Figure 6: Cost distribution of the Small database tree queries for Bushy Parallelism

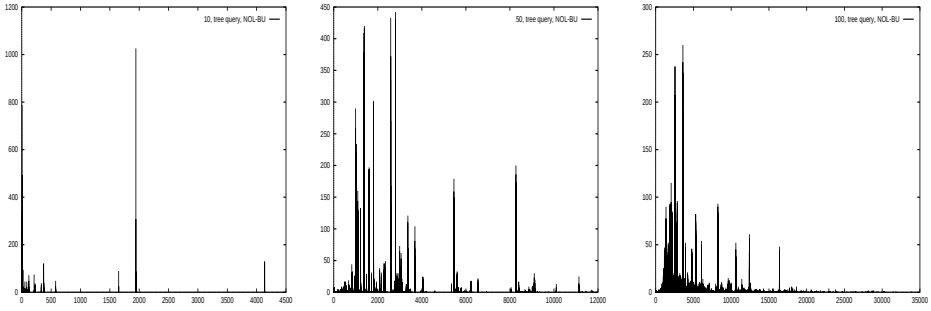

Figure 7: Cost distribution of the Large database tree queries for Bushy Parallelism

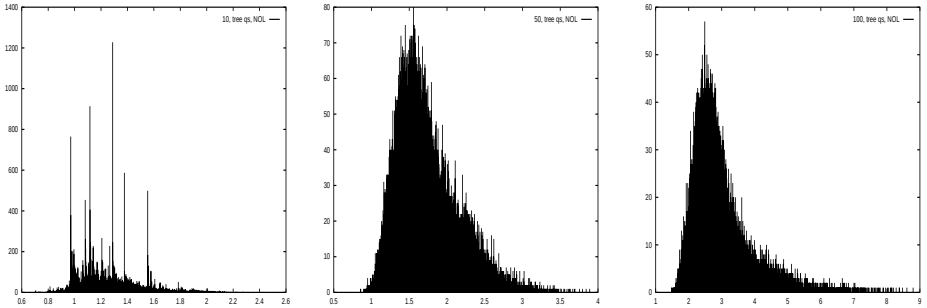

Figure 8: Cost distribution of the Small database tree queries for Bushy and Pipeline Parallelism

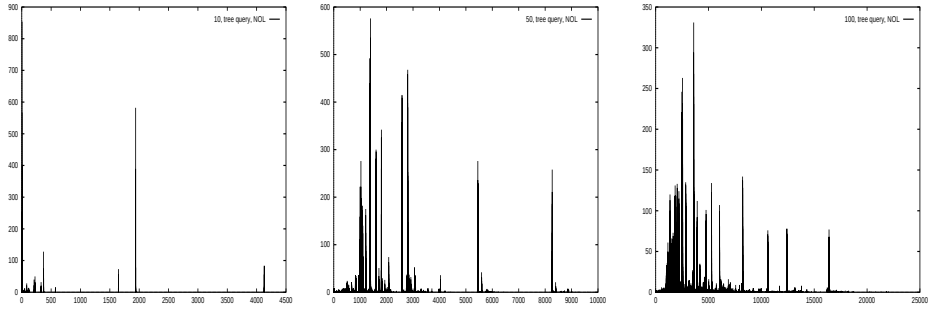

Figure 9: Cost distribution of the Large database tree queries for Bushy and Pipeline Parallelism

improvement or simulated annealing. We have however opted for a combinatorial optimization strategy as our reference technique, for the following reasons:

1. The technique described in [GLPK94] has been tested for small query sizes and sequential spaces. As the solutions space increases exponentially with the query size, and since parallel spaces are larger than sequential ones, there is no guarantee that the random method will still converge faster than a combinatorial optimization technique.

2. Our results on the cost value distribution in parallel spaces are different from those in [IK91]: High concentrations of low cost values are frequent but depend on the query graph, query size, database and cost model; γ-distributions have not been always identified. Thus, the assumption on which the random method is based [GLPK94] cannot be safely applied on parallel spaces.

3. A combinatorial optimization technique is mathematically proven to converge, while a technique randomly generating QEPs requires itself a reference technique of known behaviour to guarantee the quality of the results.

Having decided for a combinatorial optimization technique, we have chosen the parallel iterative improvement variation proposed in [SHC96]. The main characteristics of this variation are briefly as follows: A number of local runs is performed, each one resulting in a local minimum. A state (QEP) is declared as local minimum if as many transformations as the query size ("sequence length") fail to reduce its cost. Two transformations are considered, the change of the execution algorithm for an arbitrarily selected node, and the exchange between two not necessarily consecutive join-nodes, one being the ancestor of the other. The termination criterion is the expiration of a time span, set proportional to the cube of the query size.

Iterative improvement, denoted as II for short hereafter was selected for two reasons:

- It is widely tested, shown to converge fast and to produce results of satisfactory quality [SG88, INSS92, SHC96].

- It is appropriate for comparisons for which the execution time or the number of QEPs considered are predefined. Techniques like simulated annealing are less suitable for such experiments [IK90].

Combinatorial optimization techniques have been occasionally combined with augmentation heuristics specifying start QEPs of good quality [Swa89]. The usage for such heuristics to build the first generation of our GP technique is an interesting future issue. For the purposes of this study, the random generation of the start QEPs is adequate: iterative improvement will anyway converge, though slower.

5.2 Preparatory Experiments.

Before comparing our technique to iterative improvement, we need to specify a termination criterion for both strategies. Furthermore, as any other non-exhaustive strategy, GP requires tuning. Preparatory experiments were conducted to this purpose.

Parameter settings. The settings of the parallel architecture are shown in Table 2, while the database parameters are summarized in Table 1. The Small database covers databases of modest size, while the Large one covers larger databases with considerable variations in the size of their relations. Although both database types appear, we believe that the Large database is closer to real settings. However, the Small database permits experiments without the noise caused by the large relation size differences.

Termination criterion. Our comparisons concern effectiveness, in terms of quality of the optimal QEP, and efficiency, in terms of processing overhead. We did not measure efficiency as optimization time, because we wanted to eliminate the impact of implementation differences. We rather specify as termination criterion the generation of a given number of QEPs, and we compare the behaviour of the two techniques for this number of QEPs.

For the specification of the number of QEPs being considered, we have used the convergence results of II. In particular, we have let II optimize the queries for a large number of QEPs and drawn the curves describing the evolution of the optimal cost with the number of QEPs produced thus far. Those curves are shown for the different cost models and the Small and Large databases in the Fig. 12 to 17. The curves converge asymptotically to the horizontal axis. For each query, we have identified the "threshold point": the point at which the steepness of the curve, and thus the speed of convergence, starts diminishing. We call the number of tested QEPs corresponding to the threshold point of a query its "threshold number".

The threshold numbers observed range between 2,000 and 25,000 QEPs, varying with the query size and depending on the database and the cost model. We have used the threshold numbers as termination criterion for the GP technique, thus forcing our technique to compete with the already converged reference strategy.

Tuning the GP strategy. Like any non-exhaustive search strategy, GP requires tuning. One tuning parameter is the crossover/mutation ratio. In our experiments, we used fitness proportionate selection to choose 5% of the individuals for mutation and 95% for mating with crossover.

Two critical GP parameters are the number of individuals in the population and the number of generations. In general, increasing either parameter leads to better results. However, the total number of QEPs being tested is limited. Using the threshold numbers obtained from the II convergence experiments, we have tested GP with various population sizes for the 30- to 70-joins queries. We have considered the population sizes $4 \cdot n$, $7 \cdot n$, $10 \cdot n$, $13 \cdot n$, $16 \cdot n$, where n is the query size. Given the threshold number and a population size, the number of generations was computed.

The results for each query of the experiment and for each cost model are shown in Fig. 18 to 23. The horizontal axis shows the different population sizes. The vertical axis shows the QEP cost. The horizontal line in the figures displays the value of the optimal QEP produced by the iterative improvement reference technique.

By cross-checking those figures with the cost distributions presented in subsection 4.5, we observe that the QEP cost values for both II and our GP technique appear at the leftmost side of the horizontal axis. This means that GP does anyway converge to QEPs of good quality, and tuning concerns only the *fine* performance differences between the two techniques.

The evolution of the optimal cost with the population size does not converge as the population size increases. There is an interval of population sizes yielding good optima. Outside this interval, the performance degrades. This indicates that there are threshold values for both the population size and the number of generations, which may not be violated: At the left side of the horizontal axis, the population size is too small to produce individuals of good quality. At the right side, the population size is very large, but the number of generations is too small for the technique to converge to good QEPs.

The values of the optimal QEPs found by GP are generally better for the cost model considering bushy parallelism than for the one also considering pipeline. In the latter case, the optimal QEPs produced by GP are inferior to the II optimal QEP. This subject is further discussed in the next subsection, where we compare GP to II for all queries of the test suite.

Our experiment provided us with the most appropriate population size per query of the test. Using these size values and estimating the threshold values for population size and number of generations per query size, we specified the population sizes for all queries in the 10 to 100-joins query range, for both the Small and the Large databases and for the cost models expressing different types of parallelism. The tuned population size is in most cases $10 \cdot n$, where n is the query size. The number of generations is computed by dividing the threshold number for the query by the population size.

5.3 Experimental Results

Convergence of GP. In Figure 24, we show the convergence of the individuals and generations in our technique for the solutions space of the BO cost model for the 70-joins query in both databases. The x-axis shows the ranges of cost values of the produced QEPs, and the z-axis the number of individuals

per cost range. This cost distribution changes from a generation to the next, as GP converges towards lower cost values. The y-axis enumerates the generations. The evolution of cost values through the generations is displayed from the back (first generation) to the front of the picture. The gridline for the first generation roughly reflects the cost distribution of QEPs.

We observe some initial peaks in high cost ranges for the Large database. They quickly disappear, as the corresponding individuals do not survive in later generations. Rather, individuals migrate from high to low cost values. Low cost QEPs are located on the right side. The individuals gather soon at this side. The whole population finally converges towards the peak at the rightmost corner on the picture.

Comparative results. We compare the performance of our GP technique to that of iterative improvement, which we use as reference technique. Our results for the 10 query sizes and the two types of parallelism expressed by the different cost models are presented in Figure 25 for the Small database queries and in Figure 26 for the Large database queries. The horizontal axis shows the query sizes, and the vertical axis the cost values.

Since GP has been tuned according to the optimal results of II, we are comparing the results of GP to the optimal QEPs of the already converged II. Despite this bias, GP outperforms II in most cases. As can be seen from the value ranges in the vertical axis, the quality of the optimal QEPs is very high: By comparing those values to the cost distributions in Fig. 6 and 11, we see that the optimal QEPs are located in the leftmost cost value range.

When only bushy parallelism is considered, GP is always superior to II. By definition of its operators, the GP strategy encourages the formulation of bushy QEPs: by integrating subtrees to form new trees, it endeavours the increase of "bushiness" and hence, of the exploitation of bushy parallelism.

II exploits rather the pipeline characteristics of QEPs by exchanging nodes belonging to the same pipe [SHC96]. Hence, when pipelining is also considered, the performance of II improves, especially for large query sizes. For the largest queries in the Large database, II outperforms GP. This was also observed in the tuning results shown in Fig. 20 and 21 for the B&P cost model: the optimal QEPs found for various population sizes were inferior to the optimal QEP found by II.

Apparently, for large queries longer pipes can be formed and more transformations across a pipe are possible. The crossover operator of GP rather breaks those long pipes by exchanging subtrees between the mating individuals. We stress here the fact that our pipelines show no latency [HM94]; considering non-perfect pipelines [GGW95] may change this picture in favour of GP, since long pipelines would have poorer performance.

More comparative results. The above experiments have been performed for arbitrary tree queries. We have repeated the experimentation suite of II convergence, GP tuning and comparative performance for chain queries. The solutions space of chain queries is smaller than the one for arbitrary trees, since a relation participates in at most two joins only.

Our results for the 10 query sizes and the two types of parallelism expressed by the different cost models are presented in Figure 27 for chain queries in the Small and the Large database. As above, the horizontal axis shows the query sizes, and the vertical axis the cost values.

As can be seen in the Figures, the two techniques perform almost identically for the Main Memory model. When only bushy parallelism is considered, GP outperforms II in most cases, while II performs better when also pipelining is taken into account. As for the tree queries, GP is more efficient in the exploitation of bushy parallelism.

6 Conclusions

In this study, we have presented a new optimization technique based on genetic programming. Our technique uses a natural and efficient representation of the query optimization problem, conforming to the principles of genetic programming. Using this representation, we have implemented the genetic programming operators for the solutions space of valid QEPs for parallel queries.

We have compared our model to a combinatorial optimization strategy, organizing the experiments so, as to guarantee the convergence of the reference strategy. Our experiments show that GP outperforms iterative improvement, despite the bias to the reference strategy. They further show that the performance of the search strategy is affected by the types of parallelism being exploited and by the cost function modelling them: Our GP technique endeavours bushy parallelism and is thus very suitable when bushy parallelism is being exploited. Our results indicate that, in general, the characteristics of the parallel solutions space and of the cost function modelling them affect the behaviour of the search strategy. Thus, they have to be taken into account when choosing the search strategy to incorporate into the optimizer.

We are currently extending the study of the parallel spaces to cover non-perfect pipelines, as suggested in [WA91, GGW95], and to incorporate scheduling information into the cost function and the transformation moves. To this purpose, we have developed a cost model for schedule cost evaluation [SF96] and we are going to experiment with the GP technique in its solutions space. We further plan to measure the robustness and effectiveness of GP in comparison with heuristics for optimal QEP and schedule generation, as proposed in [SMK93, HM94, GI96].

References

[BFI91] Kristin Bennett, Michael C. Ferris, and Yannis Ioannidis. A genetic algorithm for database query optimization. Technical Report TR1004, University of Wisconsin, Madison, WI, 1991.

[CM95] Sophie Cluet and Guido Moerkotte. On the complexity of generating optimal left-deep processing trees with cross products. In *Fifth Int. Conf. on Database Theory – ICDT'95*, pages 54–67, Prague, Czech Republic, 1995.

[GGW95] Sumit Ganguly, Apostolos Gerasoulis, and Weining Wang. Partitioning pipelines with communication costs. In *Int. Conf. on Information Systems and Management of Data*, pages 302–320, Bombay, India, 1995.

[GI96] Minos Garofalakis and Yannis Ioannidis. Multi-dimensional resource scheduling for parallel queries. In *SIGMOD Int. Conf. on Management of Data*. ACM, 1996.

[GLPK94] César Galindo-Legaria, Arjan Pellenkoft, and Martin Kersten. Fast, randomized join-order selection – why use transformations? In *Int. Conf. on Very Large Databases*, pages 85–95, Santiago, Chile, 1994.

[Gol89] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison Wesley, Reading MA, 1989.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[HM94] Waqar Hasan and Rajeev Motwani. Optimization algorithms for exploiting the parallelism-communication tradeoff in pipelined parallelism. In *Int. Conf. on Very Large Databases*, pages 36–47, Santiago, Chile, 1994.

[Hol75] J. Holland. *Adaption in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, Michigan, 1975.

[IK90] Yannis Ioannidis and Y.C. Kang. Randomized algorithms for optimizing large join queries. In *SIGMOD Int. Conf. on Management of Data*, pages 312–321, Atlantic City, NJ, 1990. ACM.

[IK91] Yannis Ioannidis and Y.C. Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications on query optimization. In *SIGMOD Int. Conf. on Management of Data*, pages 168–177, Denver, CO, 1991. ACM.

[INSS92] Yannis Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. Parametric query optimisation. In *Int. Conf. on Very Large Databases*, pages 103–114, Vancouver, Canada, 1992.

[Koz91] John R. Koza. *Genetic Programming.* The MIT Press, Cambridge, Massachusetts, 1991.

[L+91] Rosana Lanzelotte et al. Optimization of nonrecursive queries in OODBSs. In *Second Int. Conf. DOOD'91*, pages 1–21, Munich, Germany, 1991.

[LOY94] E.T. Lin, E.R. Omiecinski, and S. Yalamanchili. Large join optimization on a hypercube multiprocessor. *IEEE Trans. on Knowledge and Data Engineering*, 6(2):304–315, 1994.

[LVZ93] Rosana Lanzelotte, Patrick Valduriez, and Mohamed Zaït. On the effectiveness of optimization search strategies for parallel execution spaces. In *Int. Conf. on Very Large Databases*, pages 493–504, Dublin, Ireland, 1993.

[Mic94] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer Verlag, second edition, 1994.

[SF96] Myra Spiliopoulou and Johann Christoph Freytag. Modelling resource utilization in pipelined query execution. In *Euro-Par'96*, Lyon, France, 1996. to appear.

[SG88] Arun Swami and Anoop Gupta. Optimization of large join queries. In *SIGMOD Int. Conf. on Management of Data*, pages 8–17, Chicago,IL, 1988. ACM.

[SHC96] Myra Spiliopoulou, Michalis Hatzopoulos, and Yannis Cotronis. Parallel optimization of large join queries with set operators and aggregates in a parallel environment supporting pipeline. *IEEE Trans. on Knowledge and Data Engineering*, 1996. To appear.

[SMK93] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Optimizing join orders. Technical Report MIP9307, Faculty of Mathematic, University of Passau, Passau, Germany, 1993.

[Swa89] Arun Swami. Optimization of large join queries: Combining heuristics and combinatorial techniques. In *SIGMOD Int. Conf. on Management of Data*, pages 367–376, Portland,Oregon, 1989. ACM.

[WA91] Anita N. Wilschut and Peter M.G. Apers. Dataflow query execution in a parallel main-memory environment. In *First Int. Conf. on Parallel and Distributed Information Systems*, pages 68–77. IEEE, 1991.

A Theorems and Proofs

Definition 6: A "subquery-tree" for a query with query graph QG is a query tree corresponding to a connected subgraph of QG.

This definition is the equivalent of a subtree for a query tree. Our aim is to introduce the notion of tree for a subquery, independently of trees for the whole query. Obviously, a subtree of a tree is also a subquery-tree.

Definition 7: We define the following functions:

- $relations(S)$ is the set of relations referenced in a subtree S.
- $nodes(S)$ is the set of join nodes in a subtree S.
- $root(S)$ is the root join node of a subtree S.
- $leftChild(n)$ is the left subtree below a join node n, if any; $rightChild(n)$ is the right subtree.
- $leftRel(n)$ is the left relation of a join node n; $rightRel(n)$ is the right relation.

Theorem 1: Let T be a valid tree. The operation $\alpha(postorder\_joinlist(T), \emptyset)$ returns a T_set consisting of valid subtrees of T.

Proof: Let $[j_1, \ldots, j_n]$ be the $postorder\_joinlist(T)$. We show that at each step of $\alpha(\cdot)$ processing a join node j_i, the T_set consists of valid subtrees of T. Whenever an ambiguity may occur, we distinguish between T_set^{old} at the beginning of a processing step and T_set^{new} output by the step.

j_1: $T\_set^{old} = \emptyset$ and $T\_set^{new} = \{j_1\}$. Since j_1 is the first node in $postorder\_joinlist(T)$, it is a leaf join of T, and as such a (trivial) valid subtree of T.

Assumption of induction: After processing the $(k-1)^{th}$-element of $postorder\_joinlist(T)$, j_{k-1}, T_set consists of valid subtrees of T.

j_k:

1. $$\nexists S \in T\_set \wedge relations(S) \cap (\{leftRel(j_k)\} \cup \{rightRel(j_k)\}) \neq \emptyset$$

   Then $\{j_k\}$ is added to T_set. We show that j_k is a leaf node in T.

   Reductio ad absurdum. j_k is not a leaf node in T, i.e. there is a subtree S of T, such that $j_k = root(S)$ and there is at least one child j_c of j_k. Without loss of generality, let j_c be the left child of j_k, so that j_c references the $leftRel(j_k)$.

   Since j_c is a child of j_k, it appears before j_k in the $postorder\_joinlist(T)$. Therefore:

   $$\exists S_c \in T\_set \wedge j_c \in nodes(S_c)$$

   By definition of $\alpha(\cdot)$, this implies that:

   $$\{leftRel(j_c)\} \cup \{rightRel(j_c)\} \subseteq relations(S_c) \Rightarrow leftRel(j_k) \in relations(S_c) \dashv Absurd$$

2. $$\exists S \in T\_set \wedge relations(S) \cap (\{leftRel(j_k)\} \cup \{rightRel(j_k)\}) \neq \emptyset$$

   Without loss of generality, we assume that S references the left relation of j_k. Then, by definition of $\alpha(\cdot)$, S becomes the left child of j_k forming a new subtree S'. S is removed from T_set and S' is added to T_set. We show that S' is a valid subtree of T.

   Reductio ad absurdum. S' is not a subtree of T. According to the assumption of the induction, S *is* a subtree of T. Obviously, $S \neq T$, otherwise S would contain j_k. Hence:

   $$\exists j_r \in nodes(T) \wedge j_r \neq j_k \wedge (S = leftChild(j_r) \vee S = rightChild(j_r))$$

- j_r appears before j_k in $postorder\_joinlist(T)$, i.e. j_r has already been processed. Then, the nodes in the subtrees below it have already been processed as well. By definition of $\alpha(\cdot)$:

  $$\exists S'' \in T\_set \wedge j_r \in nodes(S'')$$

  Since j_r is the parent of S in T, S is contained in S''. By definition of $\alpha(\cdot)$, this means that S should have been removed from the T_set, being replaced by S'' (or a subtree of S''). $\dashv$ Absurd

- j_r appears after j_k in $postorder\_joinlist(T)$, i.e. j_r has not been processed yet.
  If $S = rightChild(j_r)$ and since S is a valid subtree of T, the node in $postorder\_joinlist(T)$ appearing after $root(S)$ should be j_r, by definition of the list. However, S has already been formulated before processing j_k, and j_k appears before j_r. $\dashv$ Absurd
  So, $S = leftChild(j_r) \Rightarrow leftRel(j_r) \in relations(S)$. Since S is a subtree of S', $leftRel(j_r) \in relations(S')$. By definition of $\alpha(\cdot)$, S' will become the left child of j_r when j_r is processed. Since S is a valid subtree of T and since j_r is the parent of S in T, the nodes of the sublist $[root(S), \ldots, j_{r-1}]$ belong to the right subtree below j_r in T, by definition of $postorder\_joinlist(T)$. j_k appears in $postorder\_joinlist(T)$ after the nodes in S and before j_r, so it belongs to the right subtree below j_r in T, say S''. S'' is a valid subtree of T and so is S; hence:

  $$relations(S) \cap relations(S'') = \emptyset$$

  Moreover:

  $$j_k \in nodes(S'') \Rightarrow (\{leftRel(j_k)\} \cup \{rightRel(j_k)\}) \cap relations(S'') \neq \emptyset$$

  and, since j_k is the parent of S in T_set:

  $$(\{leftRel(j_k)\} \cup \{rightRel(j_k)\}) \cap relations(S) \neq \emptyset$$

  $\dashv$ Absurd

Theorem 2: Let T be a valid tree. The operation $\alpha(postorder\_joinlist(T), \emptyset)$ returns a T_set consisting of a single valid tree T' identical to T.

Proof: As shown in Theorem 1 for the processing of the first node in $\alpha(postorder\_joinlist(T), \emptyset)$, T_set already has a member by the end of the first processing step. In subsequent steps, a T_set member is only removed when a new subtree has been constructed and added to T_set. Hence, upon completion of $\alpha(\cdot)$, T_set is not empty.

Reductio ad absurdum. Let S be a member of T_set and let there be further members of the set. By Theorem 1, they all are valid subtrees of T. Therefore:

$$\nexists X \in T\_set - \{S\} \wedge relations(S) \cap relations(X) \neq \emptyset$$

Since $QG(V, E)$ is a connected graph and T a query tree for it:

$$\exists R \in V \wedge R \in relations(S) \wedge (\exists j \in E \wedge j = (R, R') \wedge R' \notin relations(S))$$

By definition of $\alpha(\cdot)$, this implies that:

$$\begin{aligned} j \notin nodes(S) &\Rightarrow \exists X \in T\_set - \{S\} \wedge j \in nodes(X) \\ &\Rightarrow \{R, R'\} \subseteq relations(X) \\ &\Rightarrow R \in relations(X) \cap relations(S) \end{aligned}$$

$\dashv$ *Absurd*

It is proven that upon completion of $\alpha(\cdot)$, T_set has a single member T'. By Theorem 1, T' is a subtree of T. Since all nodes in T have been processed and added to the subtrees appearing in T_set, it holds that $nodes(T) = nodes(T')$. Therefore, T' is identical to T.

T' is identical to T but for occasional switches between left and right children. In our implementation, the inner and outer stream for each join-node are specified as part of the algorithm's semantics, so that commuting the left and right child of a node does not change the tree semantics and cost.

Theorem 3: Let *nodelist* be a permutation of join nodes in a valid tree T. Let T_set be a probably empty set of valid subquery-trees, such that the sets of relations referred to in them are disjunct. Then, when applying $\alpha(nodelist, T\_set)$, all subquery-trees appearing in T_set are valid and refer to disjunct relations.

Proof: Since T_set changes after the processing of each node in the *nodelist*, we distinguish between T_set^{old} before the processing of a node and T_set^{new} after the processing of the node.

- $m = 1$.

A single node n is added to T_set, i.e. $T\_set^{new} - T\_set^{old} = \{n\}$. n is a valid subquery-tree since it consists of a single join. Moreover, by definition of $\alpha()$:

$$\{n\} \in T\_set^{new} \Rightarrow \nexists S \in T\_set^{old} \wedge relations(S) \cap relations(n) \neq \emptyset$$

- **Assumption:**

$$S \in T\_set \wedge \mid nodes(S) \mid < m \Rightarrow \nexists X \in T\_set \wedge relations(S) \cap relations(X) \neq \emptyset$$

- $S \in T\_set \wedge \mid nodes(S) \mid = m$.

Let n be the root of S and S_l the left child of n, without loss of generality. Then:

$$\mid nodes(S_l) \mid < m \Rightarrow S_l \ valid \wedge (\nexists X \in T\_set \wedge relations(S_l) \cap relations(X) \neq \emptyset) \tag{6}$$

$$S_l = leftChild(n) \Rightarrow leftRel(n) \in relations(S) \Rightarrow^{Eq.6} \nexists X \in T\_set - \{S\} \wedge leftRel(n) \in relations(X) \tag{7}$$

1. If n has a right child S_r, then it is a valid subtree referencing relations referenced by no other subtree in T_set. The proof is identical to the proof for S_l. Consequently:

$$relations(S_r) \cap relations(S_l) = \emptyset$$

2. If n has no right child, then by definition of $\alpha(\cdot)$:

$$\nexists X \in T\_set \wedge rightRel(n) \in relations(X)$$

Consequently:

$$rightRel(n) \notin relations(S_l)$$

From the two cases above and from Eq. 6 and 7, it follows that S is valid and that:

$$\nexists X \in T\_set \wedge relations(S) \cap relations(X) \neq \emptyset$$

Theorem 4: Let T, T' be two valid trees for a query with graph $QG(V, E)$. Let T_set consist of valid subtrees of T'. Then, $\alpha(postorder\_joinlist(T), T\_set)$ returns a single tree for the query QG.

Proof: We denote by T_set^{init} the T_set input to $\alpha(\cdot)$ and by T_set^{final} the T_set output by $\alpha(\cdot)$.

1.

$$S_1, S_2 \in T\_set^{init} \Rightarrow relations(S_1) \cap relations(S_2) = \emptyset$$

Since a subtree of T' is a subquery-tree, Theorem 3 is applicable. Hence T_set^{final} consists of subquery-trees referring to disjunct relations.

2. By definition of $\alpha(\cdot)$ and by the assertions of this theorem:

$$X \in T\_set^{final} \Rightarrow \forall J \in nodes(X) : J \in nodes(T) \cup nodes(T') \Rightarrow \forall J \in nodes(X) : J \in V$$

By definition of $\alpha(\cdot)$, a node J of T is either attached to T_set or already appears there (note that T_set^{init} is not empty). In either case:

$$J \in E \Rightarrow J \in nodes(T) \Rightarrow \exists X \in T\_set^{final} \wedge J \in nodes(X)$$

Thus, all joins in the query graph $QG(V, E)$ and only them appear in subquery-trees of T_set^{final}:

$$J \in E \Leftrightarrow \exists X \in T\_set^{final} \wedge J \in nodes(X)$$

Therefore:

$$r \in V \Leftrightarrow \exists X \in T\_set^{final} \wedge r \in relations(X)$$

3. T_set^{final} consists of a single tree.

$$T\_set^{init} \neq \emptyset \Rightarrow T\_set^{final} \neq \emptyset$$

Reductio ad absurdum. Let $T\_set^{final} = \{X_1, \ldots, X_k\}, k > 1$. By Theorem 3, $i \neq j \Rightarrow relations(X_i) \cap relations(X_j) = \emptyset$.

$QG(V, E)$ is a query graph and therefore connected. Hence:

$$\exists X_i, X_j \wedge X_i \neq X_j \wedge \exists J \in E \wedge leftRel(J) \in relations(X_i) \wedge rightRel(J) \in relations(X_j)$$

$$J \in E \Rightarrow J \in nodes(T) \Rightarrow \exists X \in T\_set^{final} \wedge J \in nodes(X) \Rightarrow leftRel(J) \cap rightRel(J) \subseteq relations(X)$$

This means that:

$$relations(X) \cap relations(X_i) \neq \emptyset \vee relations(X) \cap relations(X_j) \neq \emptyset \dashv Absurd.$$

Lemma 1: Crossover produces two valid trees for the query.

Proof: From Definition 5 and from Theorem theorem:crossover-prep, each invocation of the $\alpha(\cdot)$ operator in crossover produces a single valid tree for the query.

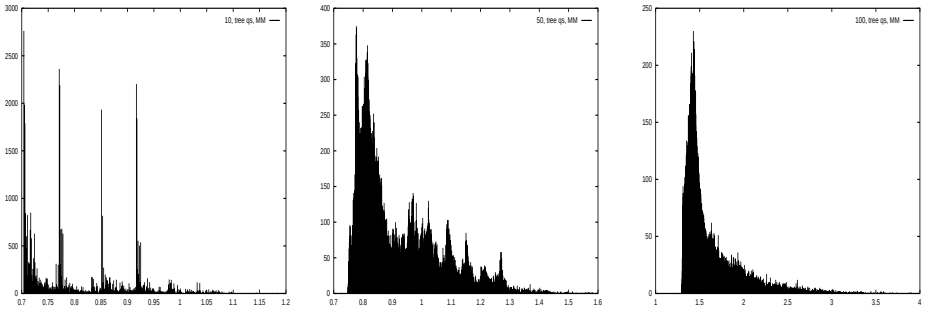

Figure 10: Cost distribution of the Small database tree queries for the Main Memory Model

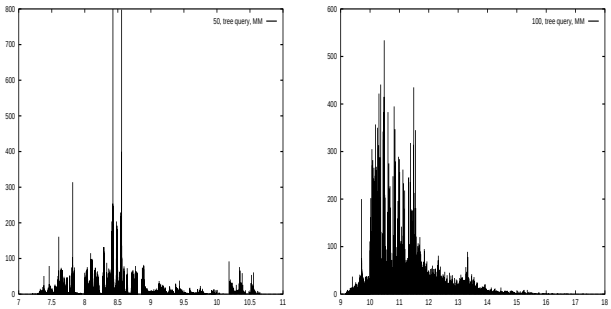

Figure 11: Cost distribution of the Large database tree queries for the Main Memory Model

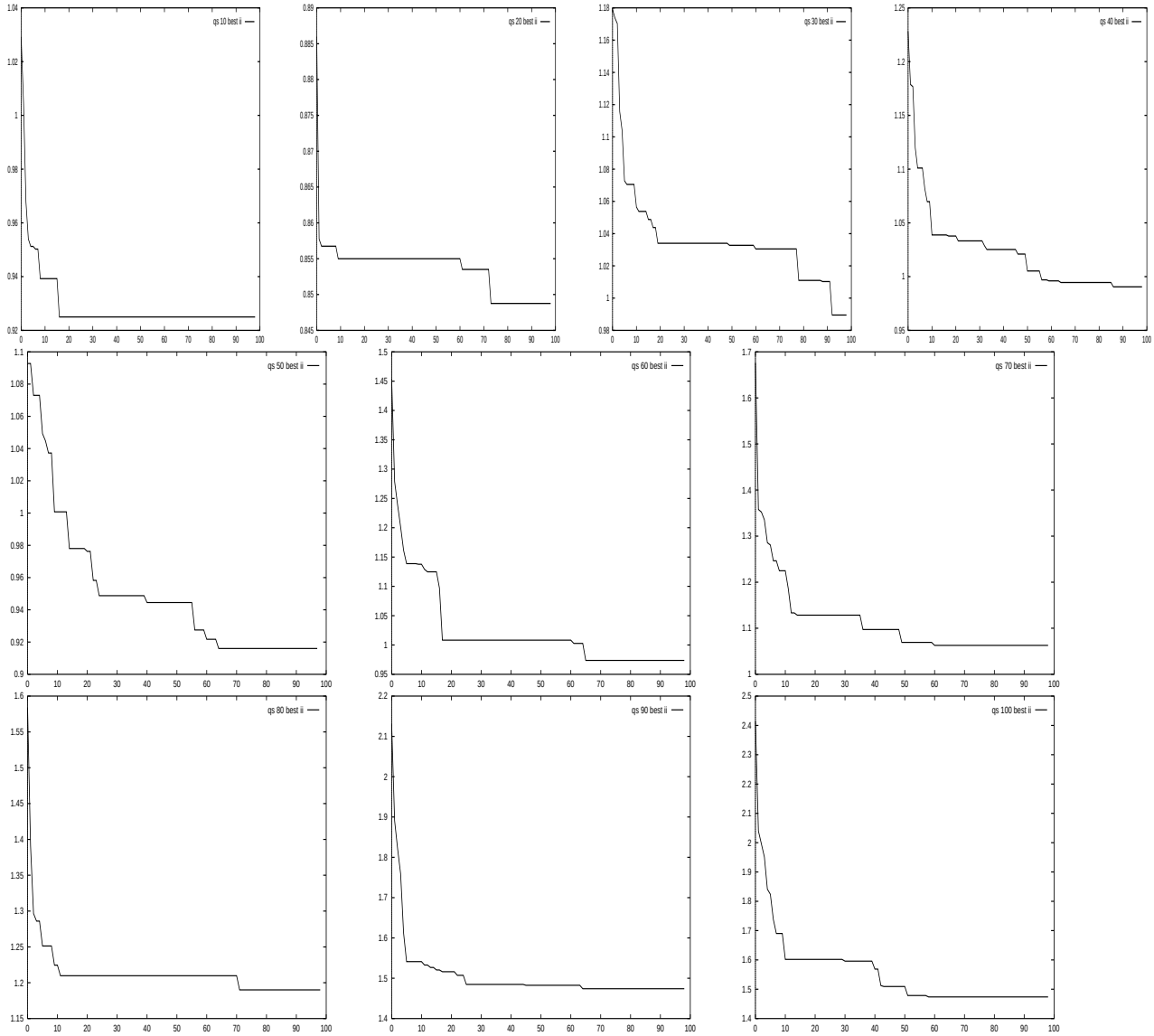

Figure 12: II convergence for tree queries in the Small database for Bushy Parallelism

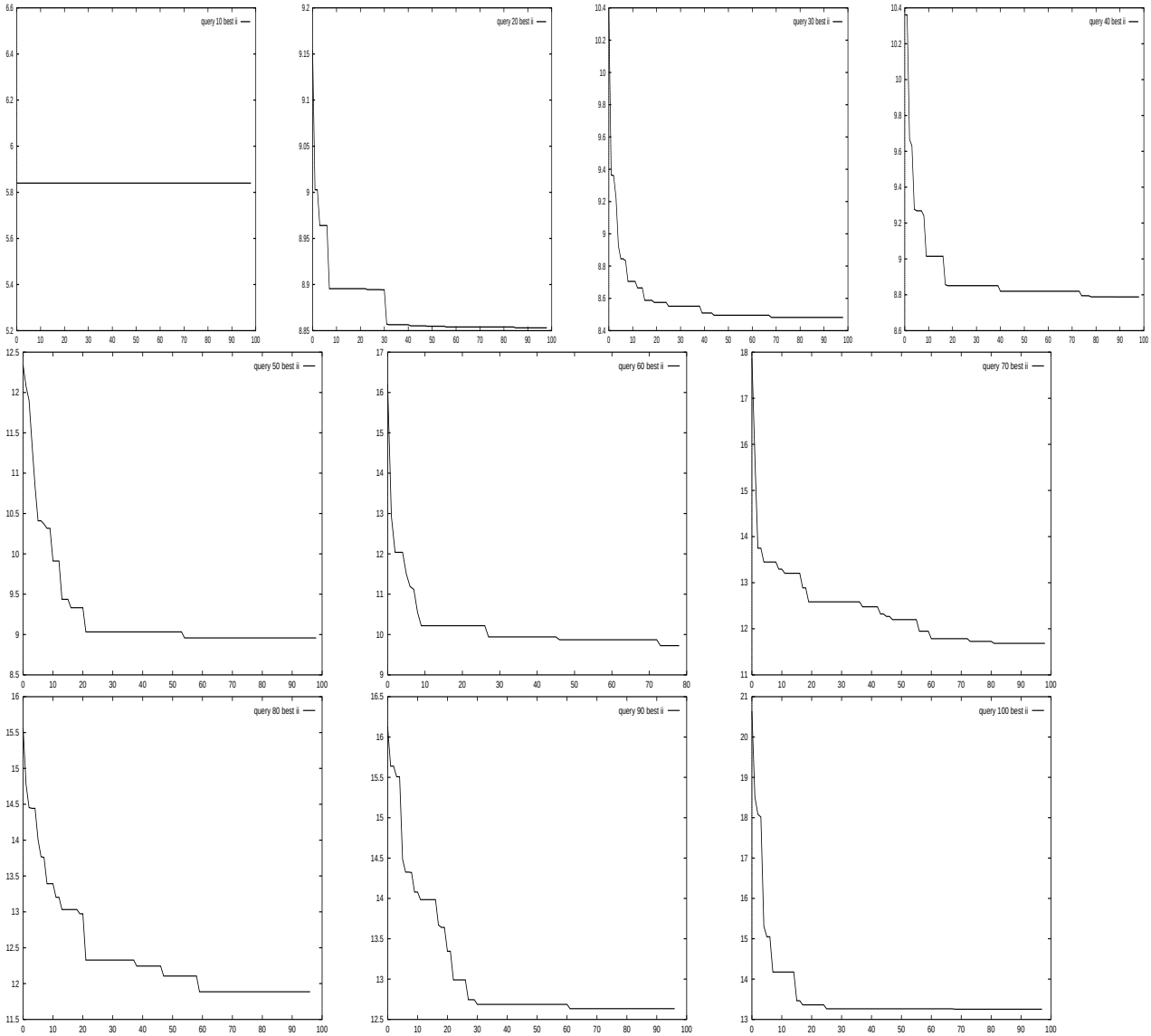

Figure 13: II convergence for tree queries in the Large database for Bushy Parallelism

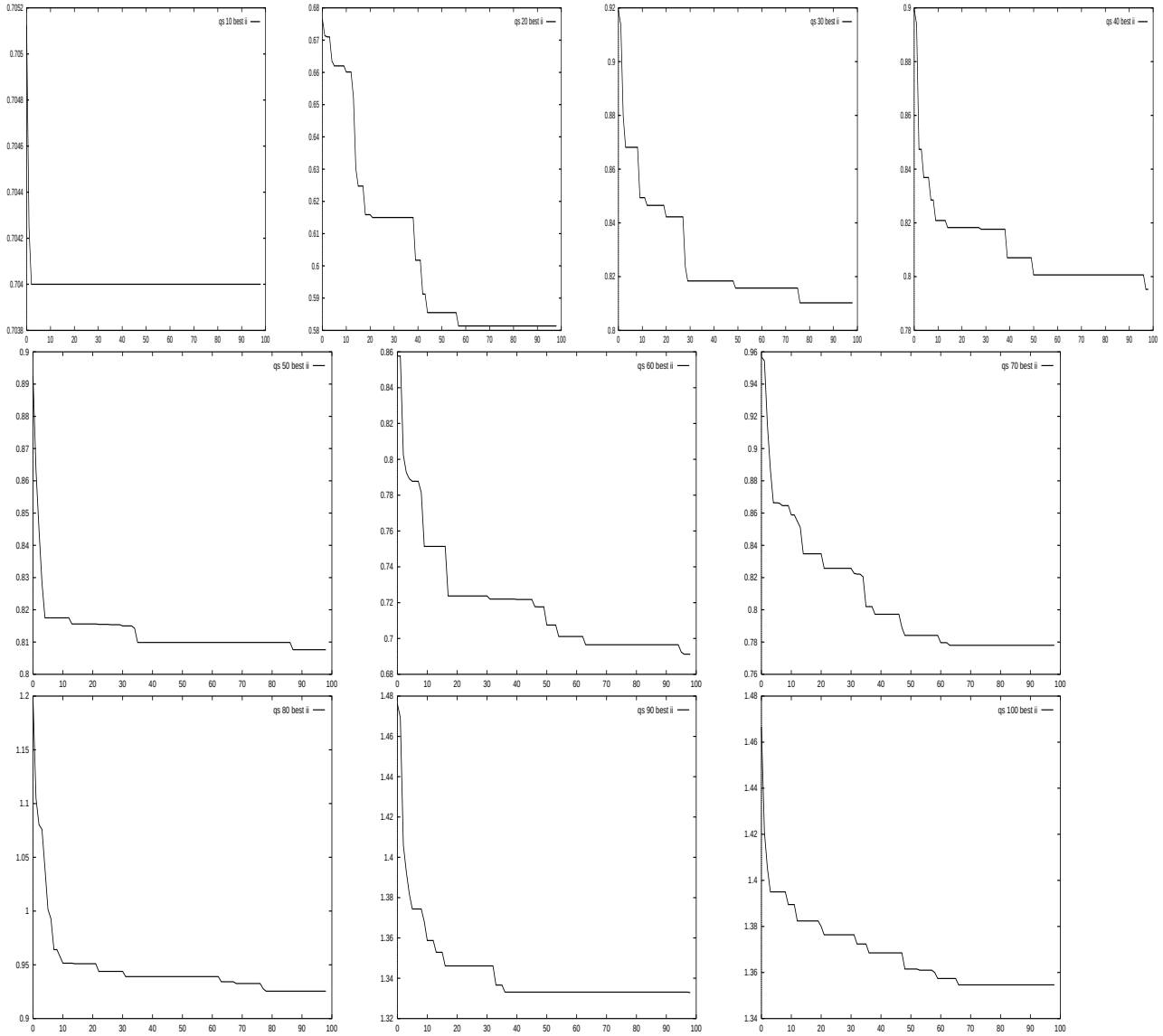

Figure 14: II convergence for tree queries in the Small database for Bushy and Pipeline Parallelism

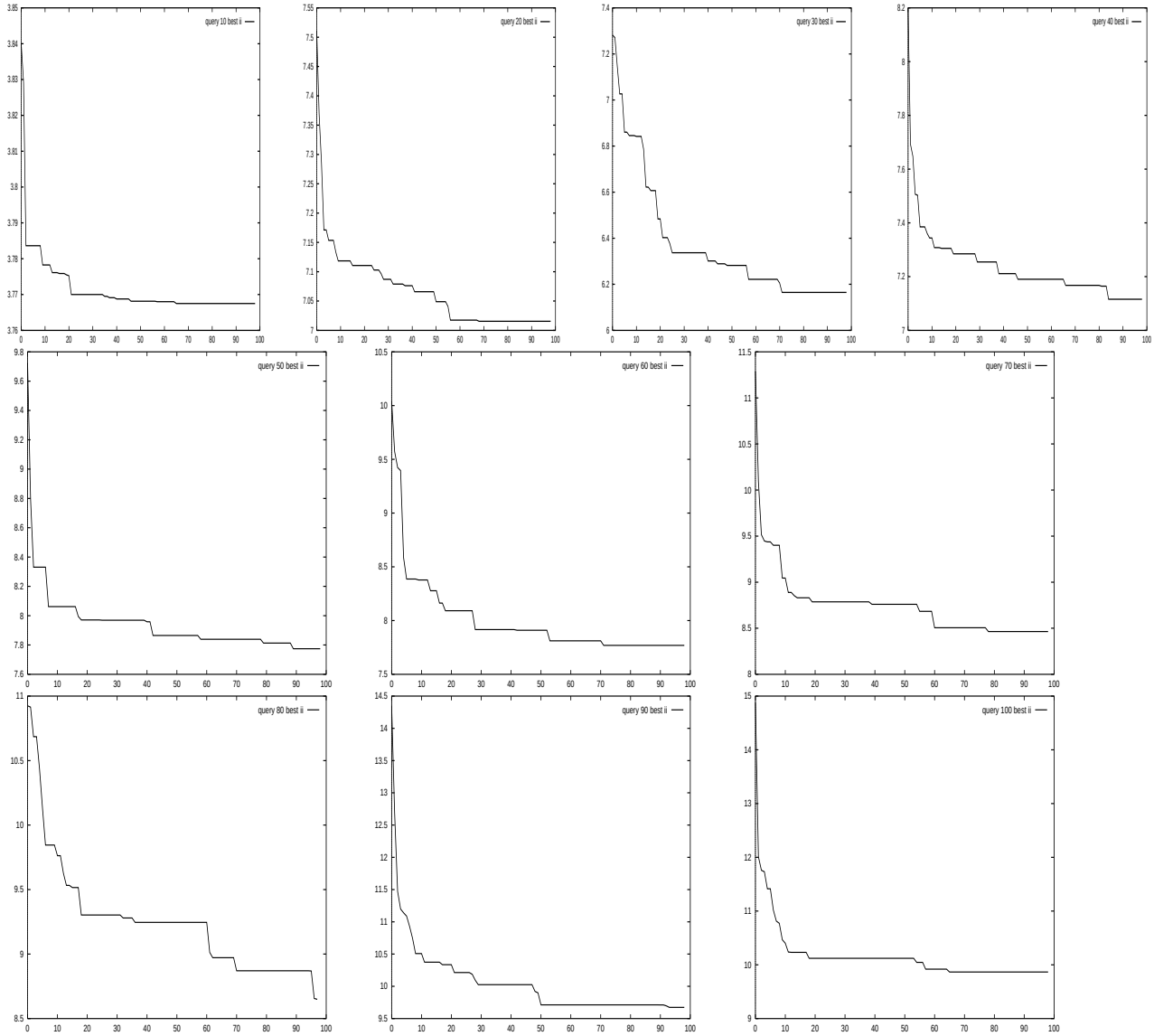

Figure 15: II convergence for tree queries in the Large database for Bushy and Pipeline Parallelism

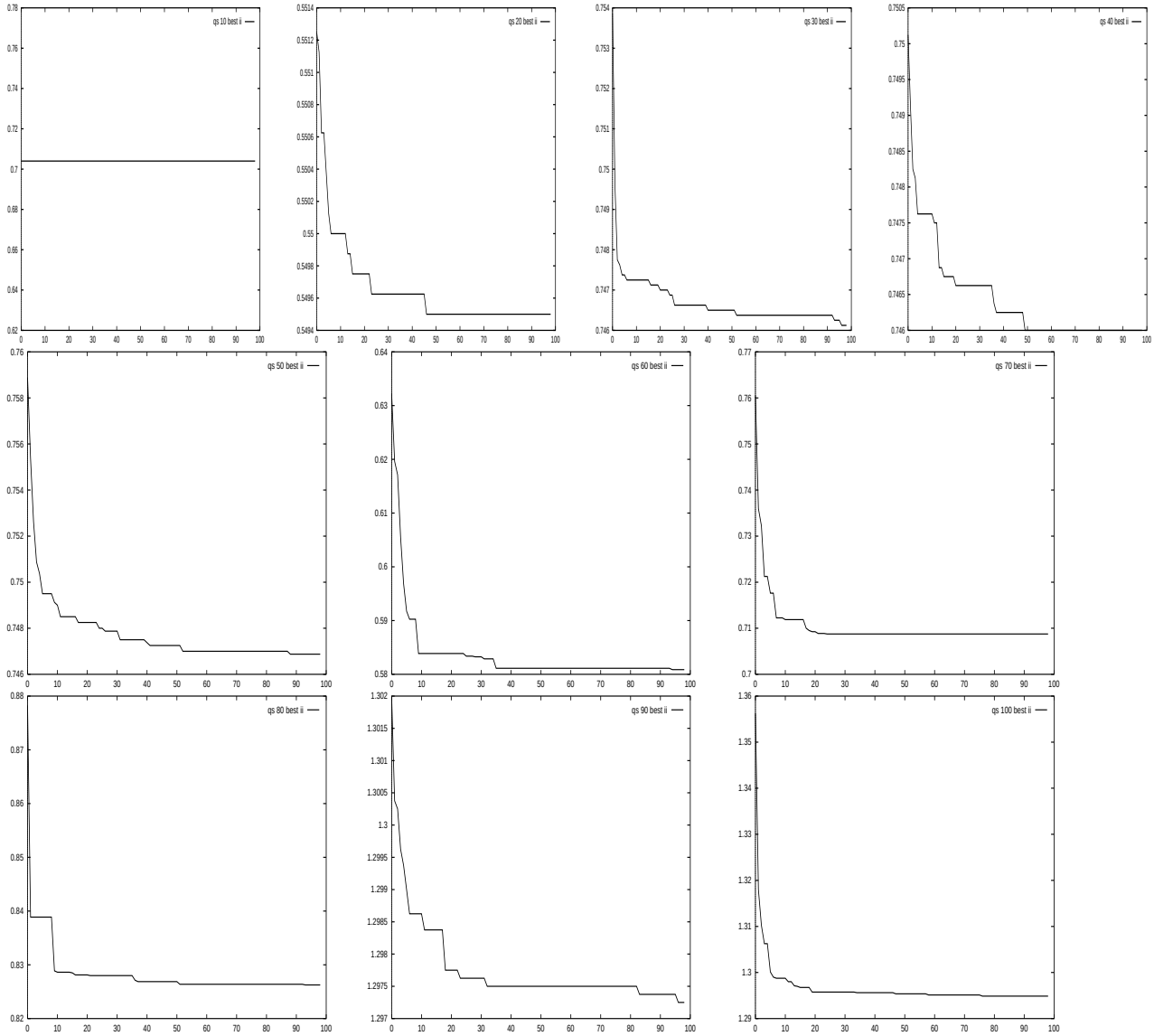

Figure 16: II convergence for tree queries in the Small database for the Main Memory Model

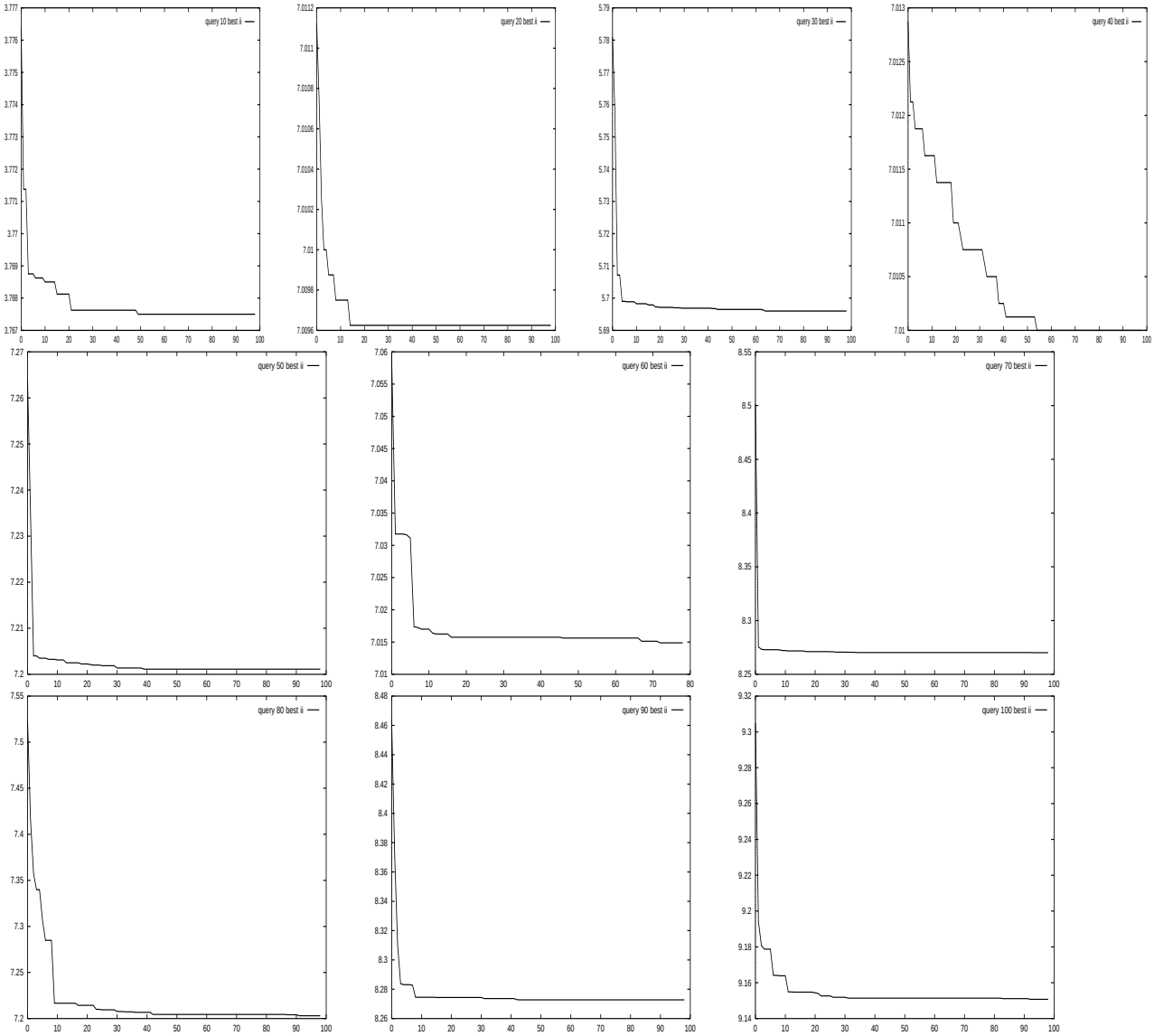

Figure 17: II convergence for tree queries in the Large database for the Main Memory Model

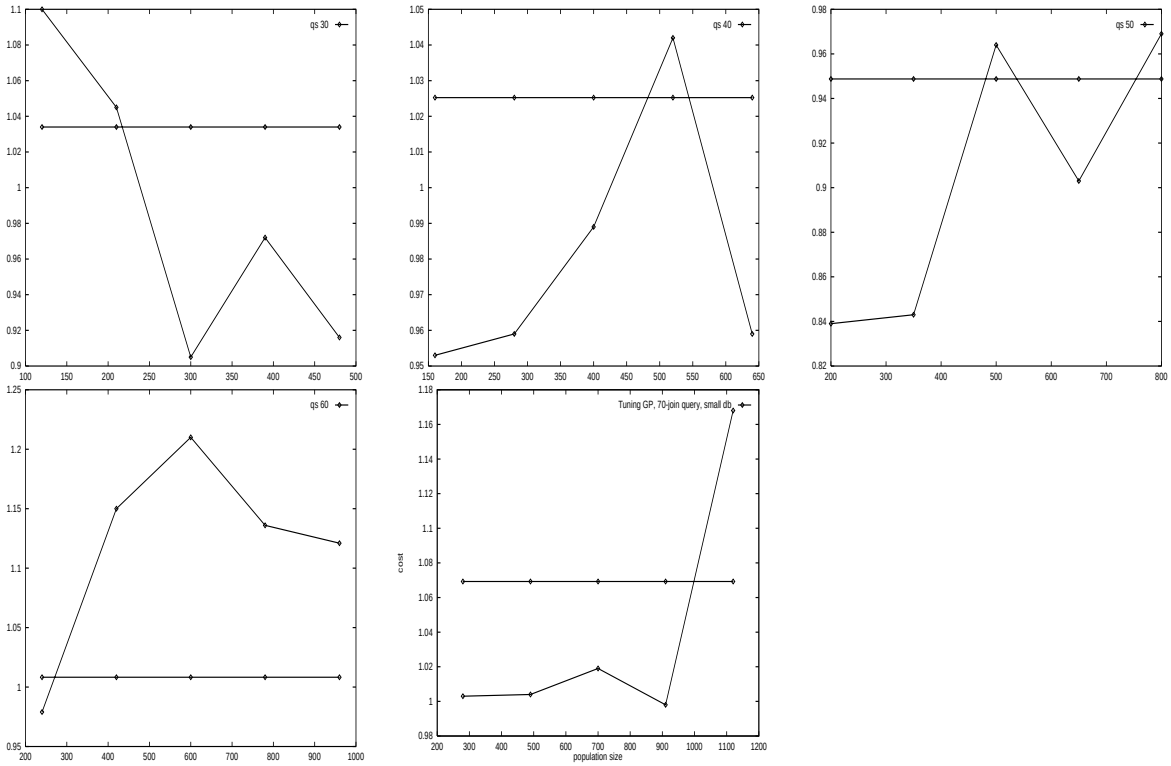

Figure 18: Tuning tree queries of the Small database for Bushy Parallelism

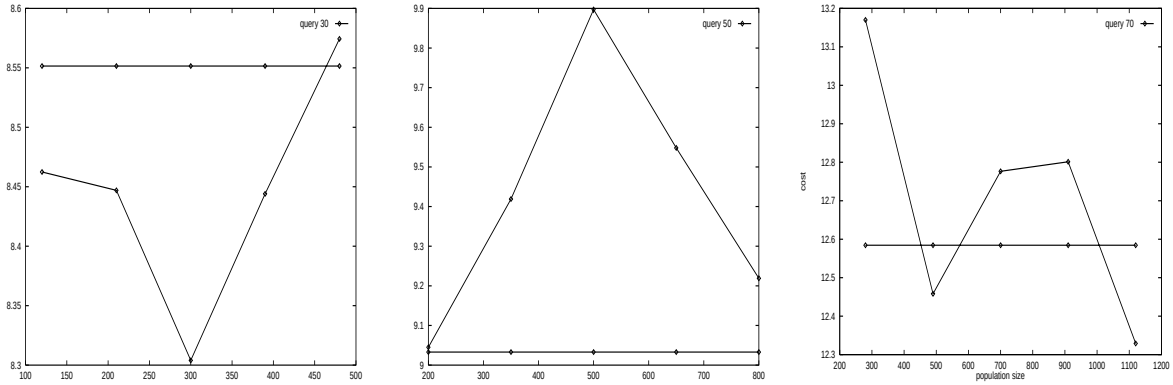

Figure 19: Tuning tree queries of the Large database for Bushy Parallelism

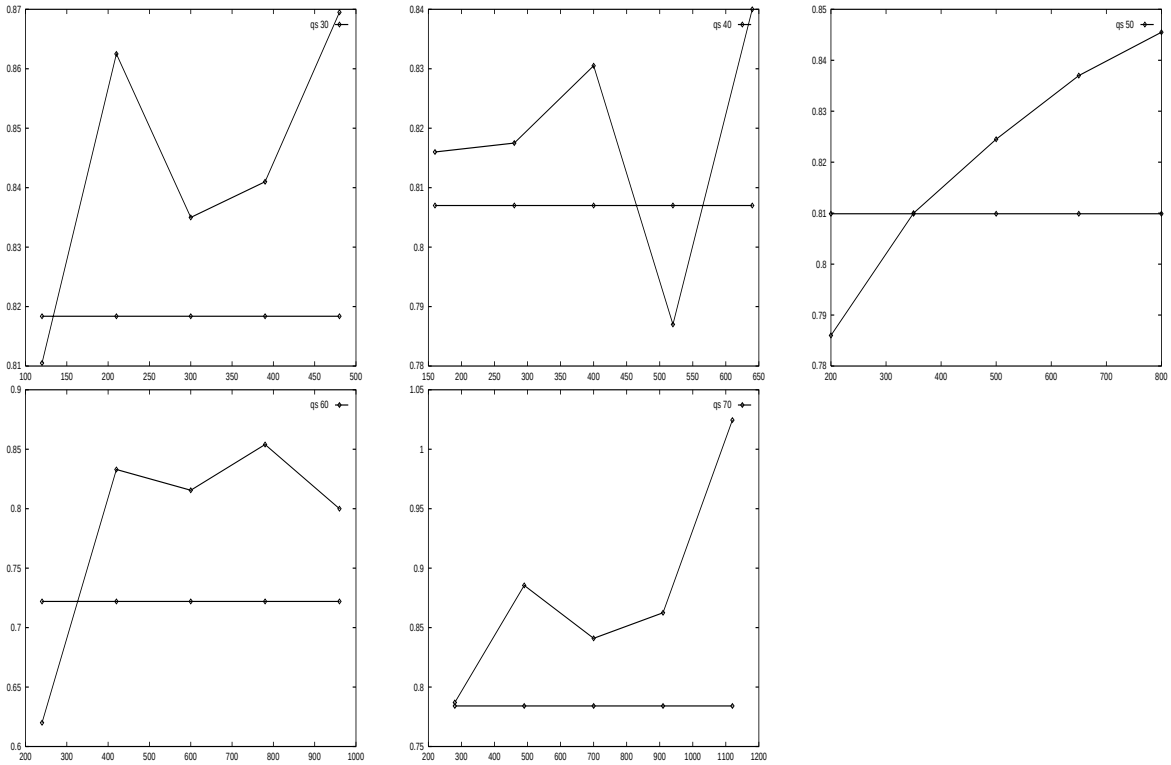

Figure 20: Tuning tree queries of the Small database for Bushy and Pipeline Parallelism

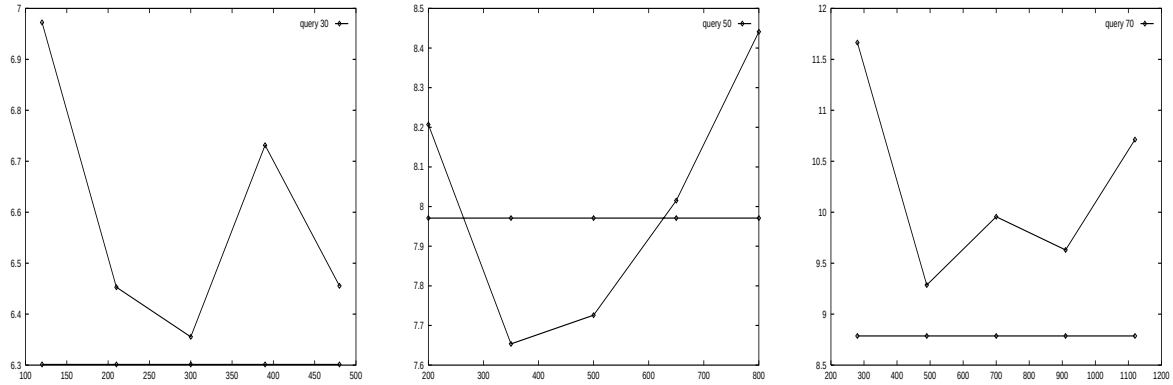

Figure 21: Tuning tree queries of the Large database for Bushy and Pipeline Parallelism

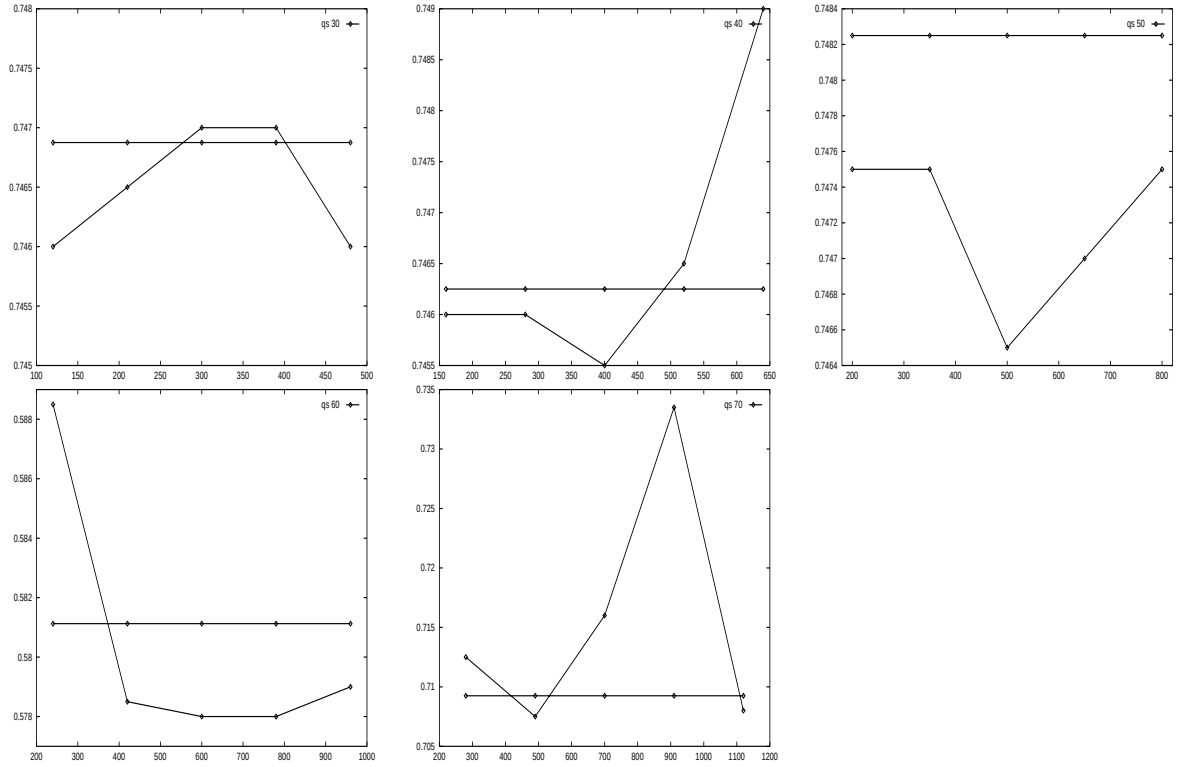

Figure 22: Tuning tree queries of the Small database for the Main Memory Model

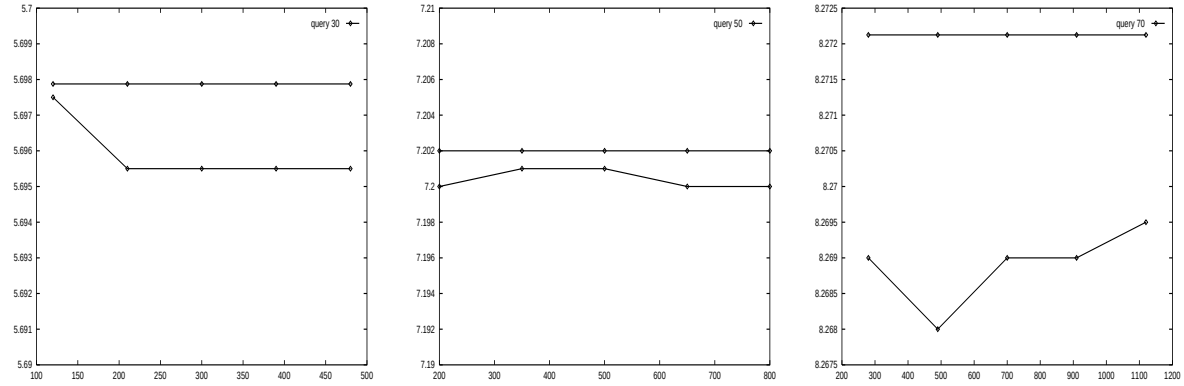

Figure 23: Tuning tree queries of the Large database for Main Memory Model

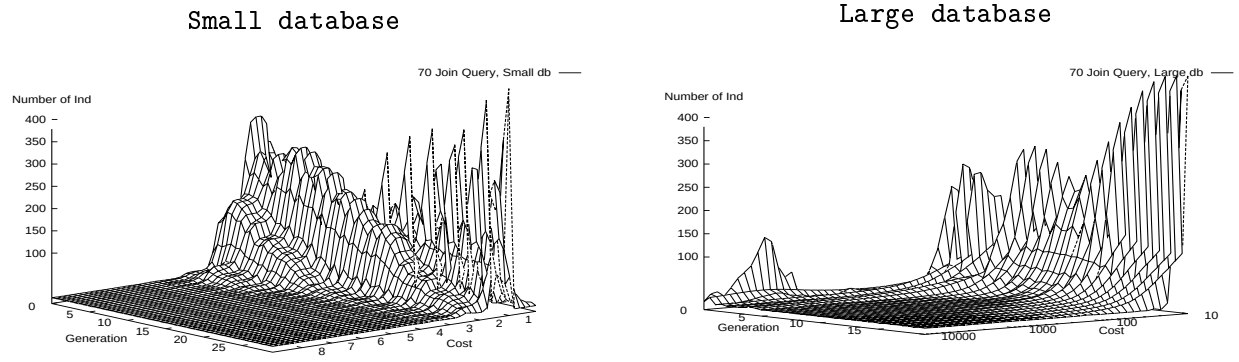

Figure 24: Evolutional convergence of the GP technique for 70-join queries exploiting bushy parallelism

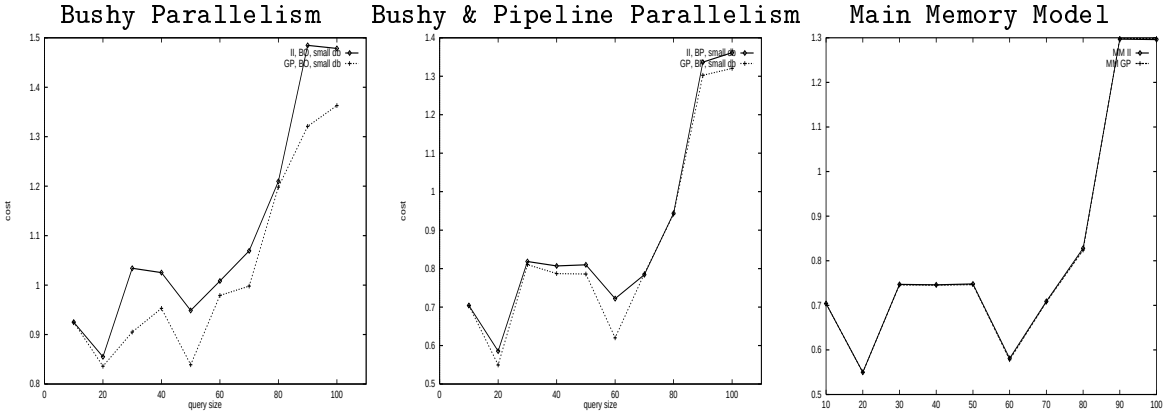

Figure 25: Cost of the optimal QEPs for the tree queries in the Small database

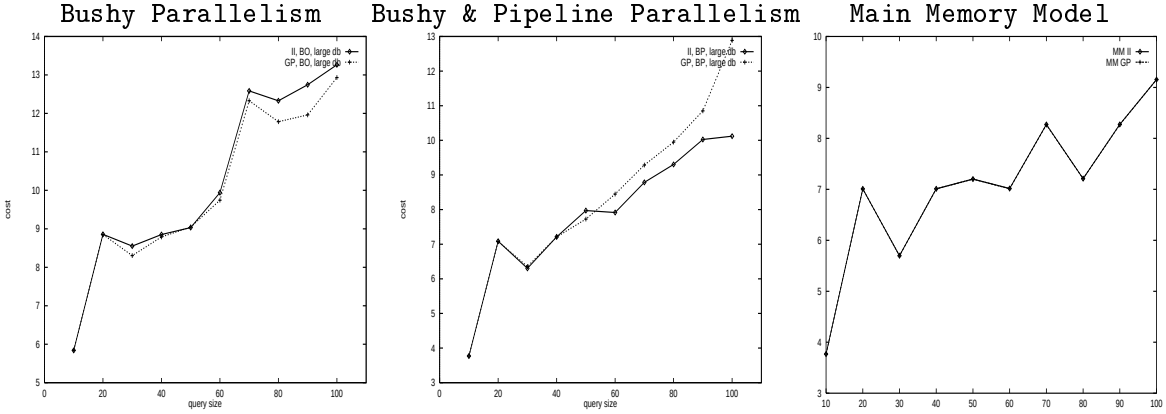

Figure 26: Cost of the optimal QEPs for the tree queries in the Large database

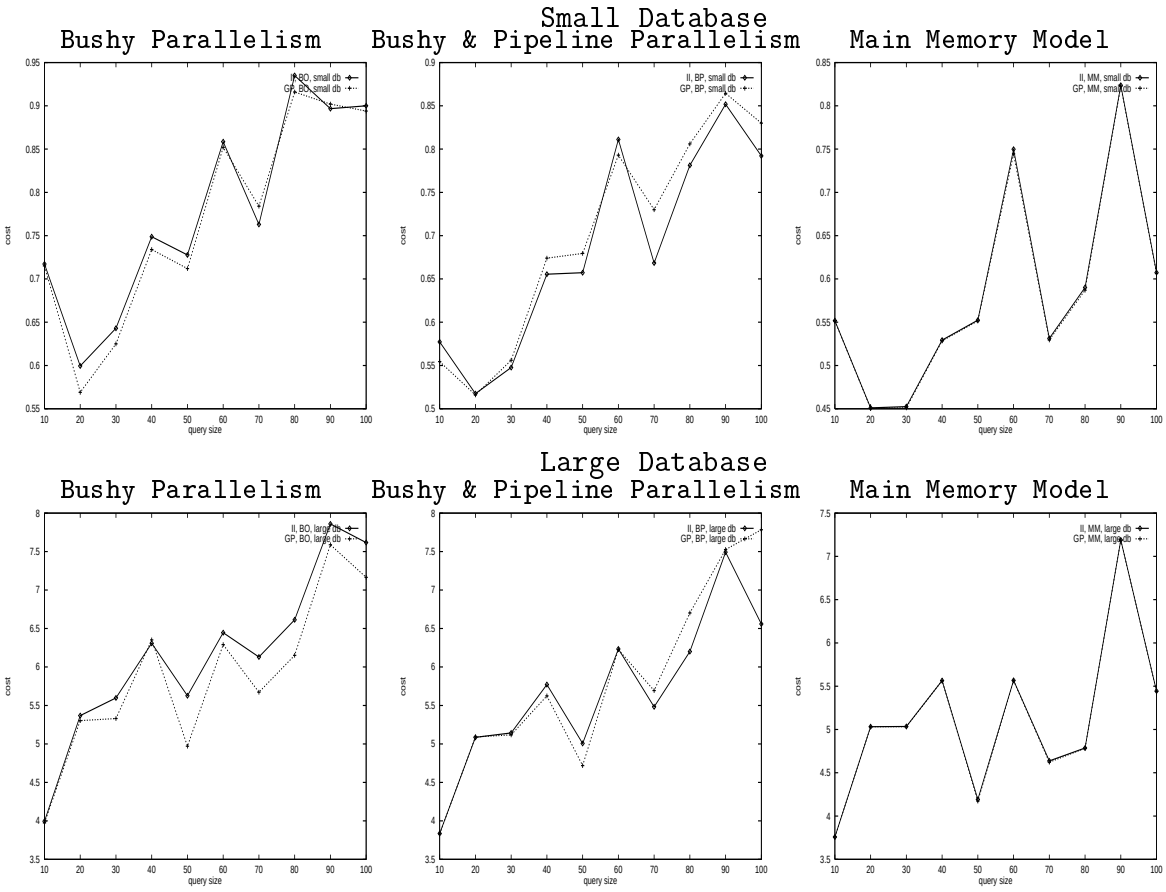

Figure 27: Cost of the optimal QEPs for the chain queries