Abstract

After over twenty years of study, query optimisation in distributed databases is still an open problem, particularly in a dynamic network environment. This study aims to apply probabilistic optimisation techniques to a system with non-uniform bandwidth between servers to investigate the feasibility of applying such techniques at run-time to accommodate changing network conditions. The cost-based optimiser developed manipulates both operator order and the physical locations at which these operators are evaluated, and provides a basis for the detailed examination of resource contention issues or the study of new operator types.

1. Introduction

Industrial-strength Database Management Systems today are generally designed to operate within a Client-Server environment. The Client-Server paradigm allows database client applications to submit queries or requests for data across a Local or Wide Area Network (LAN or WAN) to one or more database servers which return the required data over the same network. The Client-Server paradigm has also allowed the creation of Distributed Database Management Systems (DDBMS) in which data can exist as relations at diverse physical locations on autonomous servers, and these have presented new opportunities for creating high-performance fault-tolerant database systems but at the cost of increased complexity.

In many studies of query optimisation in distributed database systems [1, 2, 5], the data transfer rate of the network connection between two servers is assumed to be uniform throughout the system, independent of the servers involved. Also, none of these studies consider the possibility that query processing may be bottlenecked by lack of bandwidth on particular links.

The aim of this study is to enhance the optimisation strategy for distributed systems to account for non-uniform communication costs, particularly where the system is bottlenecked by lack of bandwidth in particular links. This study models a system with two possible bandwidths across which intermediate relations must be transferred, one corresponding to a fast link (100Mbps LAN) and one corresponding to a congested link in which page transfer times are in the same order as the page processing times of individual operators. Effective bandwidth between pairs of servers is modelled using a two-dimensional table of cost parameters. This allows the effect of non-uniform and even asymmetric bandwidth between individual servers to be investigated.

This paper is organised as follows: Section 2 discusses query processing in the context of a distributed database environment and some of the concepts behind the optimiser implemented in this study. Section 3 explains the proposed method and the overall design of the experiment conducted and Section 4 discusses the implementation of the optimiser, the results and a discussion of their significance. Section 5 concludes and presents where there is scope for further work.

2. Query Processing and Optimisation

2.1 Query, Data & Hybrid Shipping Strategies

Typically, RDBMS's utilise query shipping, in which the client (consumer) dispatches a query to the server (producer) which processes it and returns the result tuples. Query shipping greatly reduces network traffic for low-selectivity queries and, since the bulk of the processing is performed at the server, allows the use of resource-poor (and, hence, cheaper) client machines. Query shipping, however, under-utilises client resources and can produce processing bottlenecks when server resources become inadequate. Query shipping is the method used in "traditional" standard client server systems with a single, centralised server processing queries submitted by clients.

In contrast to RDBMS's, OODBMS's generally utilise data shipping, in which the client (consumer) first checks its local cache for the required data (where client-side caching is implemented) and, if necessary, requests a fresh copy of the data from the server (producer) in order to process it locally. Data shipping decentralises the processing required by fully exploiting the CPU, disk and/or memory resources of powerful client machines, allowing the system to scale up naturally as clients and their associated resources are added to the system. In data shipping, the server in effect becomes a page server, sending required data to the client in minimum units of one page.
Franklin, Jönsson & Kossmann [1] observed that neither query or data shipping can provide optimal performance under all conditions and so proposed a hybrid shipping strategy in which the DDBMS query optimiser determines where the processing is to take place at run-time. This method potentially allows a system to take into account factors such as the expected size of the data to be shipped, network traffic, and the relative performance and loading of both the client and server machines, to ensure optimality of query execution. This approach, while equalling or bettering the performance of pure query or data shipping, results in a significant increase in the number of query execution plans to be considered for optimisation (the solution space).

It should be noted that both query and data shipping are special cases of the more general hybrid shipping case. Where a hybrid shipping optimiser is constrained to perform all operations at the consumer, it uses data shipping and, conversely, where it is constrained to perform operations at a producer, it uses query shipping. The optimiser implemented in this paper is not constrained to any particular shipping, so it is free to choose solutions including or ranging between either of these two extremes.

2.2 Randomised Optimisation

The algorithm selected for use in this study is the Two-Phase Optimisation (2PO) algorithm used by both Franklin, Jönsson & Kossmann [1] and Ioannidis & Kang [4], since these studies and others have confirmed its efficacy. The 2PO algorithm is a combination of Iterative Improvement (II) and Simulated Annealing (SA) algorithms. 2PO takes advantage of the rapid convergence of II along with the quality of the solutions produced by SA [7, 4, 1].

![Figure 1. Iterative Improvement versus Simulated Annealing](image)

A detailed description of II, SA & 2PO can be found in Ioannidis & Kang [4], and optimisation parameters used in this study are virtually identical to those used in [4]. Below in Figure 1 is an illustration of the Iterative Improvement (II) and Simulated Annealing (SA) algorithms as they "wander" through the solution space towards an optimal solution by checking a random selection of neighbouring points for the lowest cost alternative [6]. SA differs from II in that it allows some "hill-climbing" (at a probability determined by the

Temperature parameter) which prevents it from being "trapped" by a local minimum. Please note that the three-dimensional surface shown below is illustrative only, since in practice there are usually many more than two degrees of freedom for the solution space.

2.3 Canonical Query Evaluation Plan

As per [1], this study examines only chained joins with moderate selectivity. A chained join query is one in which every base relation except the first and last is joined to exactly two other relations, so the relations can be viewed as being arranged in a linear chain. In such a configuration, each base relation need only be scanned once.

In any join configuration, the precedence constraints (producer-consumer dependencies) between operators determine which query execution plans are equivalent to the canonical plan. For example, one cannot perform a join on an attribute shared by two base relations A and B if neither subtree whose output is consumed by the join contains relations A or B; that is, the join operation naturally cannot be performed before the scans on its required base relations. This means that there are fewer equivalent trees for a chain join than a star join configuration in which all relations except the central relation are joined only to the central relation on a common attribute. This makes the chain join a more interesting proposition for study.

The form of a canonical query evaluation tree for a chain join for each of the left and right deep search spaces can exist in two obvious orders, e.g. (((R1 x R2) x R3) x ... x Rn) x Rn+1 x Rn+2 or (((Rn x Rn-1) x Rn-2) x ... x R3) x R2 x R1 for a left-deep tree. By definition, the canonical tree to which subsequent transformations are applied is the starting point for all subsequent moves through the solution space, so the optimiser implemented randomly selects between the two forms when the solution space is artificially restricted to either left or right deep trees. This is to avoid trapping a randomised optimisation algorithm in a particular region of the solution space by allowing it to start from either point.

3. Proposed Method

The proposed method in this study has two main components: the methods developed for join order transformations, and evaluation site transformations. Each of these methods will be discussed in the following sections.

3.1 Join Order Transformation Primitives

In order to move from one point in the search space to another, it is important to be able to transform the join order in the query evaluation plan tree in such a way as to produce an equivalent tree. In our implementation of a distributed query optimiser, we have designed two join order transformation primitives, "swap-producer" and
“move up”, which can be combined to generate all valid join order configurations and, furthermore, can be restricted to ensure that the tree remains in left or right deep form. The aim in doing this was to obtain a small number of transformations which can potentially be applied to both unary and binary operators which can be called independently of the form of the tree in which the operator participates.

The first of the elementary transformations used in this implementation is the “swap-producer” transformation, or join commutativity, which by its nature, cannot violate any dependencies between producers and consumers. This transformation is very straightforward and can always be performed in a bushy tree after such a transformation.

With the “move-up” operator to preserve the form of the solution space, however, it is only used in conjunction with the “move-up” operator to preserve the form of the tree after such a transformation.

The second and more complex of the join order transformation primitives used is the “move up” transformation. Central to this transformation is the concept of a precedence grouping with respect to a given operator. A precedence grouping is a partial subtree with one incoming arc and one outgoing arc which does not contain any operators on which a nominated superior operator depends. This allows the precedence grouping to be cut from the tree and re-inserted above the nominated operator. For example, a join and one of its associated two scans form a precedence grouping if the scan is not required by the superior operator. To perform the “move up” transformation on a given operator, the operator is tested as a precedence grouping by checking whether it (and one of its subtrees if it is a binary operator) can be moved one level up the tree towards the display operator (the root node) without violating the precedence constraints of the immediately superior operator. If the test fails, the superior operator is included in the precedence grouping which is then the subject of another attempted “move up” transformation above the next higher operator in the tree. If the display operator is reached, then the “move up” transformation obviously cannot take place.

The left deep and right deep tree forms can be preserved in a “move up” transformation by constraining the choice of subtree to be included in the precedence grouping moved upwards to be included in the precedence grouping moved upwards with the operator and/or combining the “move up” with a subsequent “swap-producer” transformation on the same operator. It should be noted that this transformation naturally cannot be applied to a leaf node, (i.e. a scan node), the root display node or any node immediately below the root display node. A pseudocode outline of the “move-up” transformation is as follows:

### Procedure Move Up (Precedence Grouping)

If top operator of precedence grouping is display, scan or anything immediately below display

Then move failed

Else

White valid move not found and there is another subtree of bottom operator to test dependencies against

do

Decide which producer subtree of bottom operator to leave in place

Check dependencies of operator immediately above top operator

If dependencies respected by move

Then

Cut precedence grouping from tree, grafting fixed subtree back in place

Insert removed precedence grouping above operator immediately above former position

Move succeeded

Else

Add operator immediately above former position to top of precedence grouping

Recursively call “move-up” with newly extended precedence grouping

Test dependencies against next subtree of bottom operator

Done

Figure 2. Pseudocode Outline of “Move-Up” Transformation

There are some refinements to the above procedure included in the implementation, particularly in the selection of the fixed subtree in binary operators or the optional application of a subsequent “swap-producer” transformation in order to preserve either left-deep or right-deep tree form if required, but the essential features are as outlined above. If desired, other heuristics can also be applied to further restrict the moves allowed, as has been done in this study by grouping associated join-select pairs.

<table>
<thead>
<tr>
<th>Case</th>
<th>Algebraic Representation of Transformation rule</th>
<th>Equivalent Transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>(A x B) x C → (A x (B x C))</td>
<td>T_R or T_R'</td>
</tr>
<tr>
<td>L2</td>
<td>(A x B) x C → (A x (C x B))</td>
<td>T_R or T_R'</td>
</tr>
<tr>
<td>L3</td>
<td>(A x B) x C → (A x (C x B))</td>
<td>T_R or T_R'</td>
</tr>
<tr>
<td>L4</td>
<td>(A x B) x C → (B x A) x C</td>
<td>T_R or T_R'</td>
</tr>
<tr>
<td>L5</td>
<td>(A x B) x C → (B x (A x C))</td>
<td>T_R or T_R'</td>
</tr>
<tr>
<td>L6</td>
<td>(A x B) x C → (B x (C x A))</td>
<td>T_R or T_R'</td>
</tr>
<tr>
<td>L7</td>
<td>(A x B) x C → (B x C x A)</td>
<td>T_R or T_R'</td>
</tr>
<tr>
<td>L8</td>
<td>(A x B) x C → (C x A) x B</td>
<td>T_R or T_R'</td>
</tr>
<tr>
<td>L9</td>
<td>(A x B) x C → (C x (A x B))</td>
<td>T_R or T_R'</td>
</tr>
<tr>
<td>L10</td>
<td>(A x B) x C → (C x (B x A))</td>
<td>T_R or T_R'</td>
</tr>
<tr>
<td>L11</td>
<td>(A x B) x C → (C x (B x A))</td>
<td>T_R or T_R'</td>
</tr>
<tr>
<td>L12</td>
<td>(A x B) x C → (C x (B x A))</td>
<td>T_R or T_R'</td>
</tr>
<tr>
<td>L13</td>
<td>(A x B) x C → (C x (B x A))</td>
<td>T_R or T_R'</td>
</tr>
<tr>
<td>L14</td>
<td>(A x B) x C → (C x (B x A))</td>
<td>T_R or T_R'</td>
</tr>
</tbody>
</table>

Table 1. Equivalent Join Order Transformation Rules for Neighbour Function

The completeness of these of transformation primitives can be demonstrated by expressing the transition to every possible join order configuration of a

820
two-join subtree in terms of the transformation primitives. Larger trees can be viewed as nested two-join subtrees in which the relations are themselves subtrees, so the results apply to any tree. The algebraic representation of each transformation and its associated equivalent(s) is shown in Table 1. The move up operator is designated \( \overrightarrow{1} \), or \( \overrightarrow{r} \) for a “move-up” transformation with left or right subtree (respectively) remaining in position. The “swap-producer” transformation is designated \( \leftrightarrow \) when it is applied to the bottom operator of a join pair, or \( \leftrightarrow \) if it is applied to the topmost join in the pair.

Note that where the move up transformation has a different annotation to the original subtree form, (e.g. Left c.f. Right) as in cases L1 or R1, then the form of the resultant subtree is changed.

3.2 Evaluation Site Transformations

In their study, [1] used a notation scheme for each node whereby each node is evaluated at the same site as one of its adjacent operators. They discuss the concept of well-formed plans in which every operator has a clear annotation path to either the root (display) node or any leaf (scan) node. This means that no cycles exist, i.e. every connected pair of operators has at least one annotation not pointing to the other member of the pair. This means declaring certain annotations invalid wherever cycles are created by a transformation. Technically, such plans which contain cycles are not invalid, but, rather, ambiguous, and so must be altered to allow deterministic resolution of the evaluation site of each operator. This annotation scheme is completely independent of the actual operator types involved and must be re-determined for the affected parts of the tree after any operator order transformation has occurred.

Annotation resolution problems are created where the tree is transformed and the new configuration of existing annotations in the relocated operators produces a cycle. The combination of the cut and insert sub-operations used in the “move-up” operation must resolve such potential cycles to maintain a consistently well-formed plan. One approach to resolving annotations would be to propagate a new annotation outward from a nominated operator by recursively following the annotation to its location source operator and changing its annotation if it produces a cycle with the operator with the previously revised annotation. This approach was deemed unsuitable for use with the randomised optimiser, since it can result in a large number of operators having their physical locations changed. Care must be taken to minimise such perturbation to the tree to ensure that the difference between the cost of the original and the revised plan, is as small as possible, allowing the optimiser to “step” through the solution space, rather than taking large “leaps” which will hinder convergence. This should be achieved by revising the annotation in a manner as consistent with the original physical locations of the individual operators as possible.

Essentially, the operator annotations should be regarded as a “means to an end” to determine the physical locations of the operators, rather than a factor directly affecting plan cost. Because of the need for efficiency in the transformations, it is not feasible to exhaustively determine which revised annotation scheme will best match the existing physical locations of operators after a join order transformation. Because of this, we adopted a rule-based approach to change individual operators’ annotations to point to the same operator they pointed to before the transformation, thereby minimising perturbation to the tree and consequently drastic cost changes. Once again, for efficiency’s sake, this rule based method of annotation resolution need not be absolutely optimal, only an average improvement over a brute-force method.

4. Performance Evaluation

4.1 Experimental Procedure

The program was designed and implemented in C++. While only being run for a query on three base relations (comprised of 8 operators) in this study, the program is dimensioned to accommodate queries on up to ten base relations (comprised of 29 operators) on up to eleven machines. This study is restricted to a configuration of one client with no local base relations with three servers each containing a single base relation, but there is no restriction imposed by the program on the potential location of base relations, allowing configurations where any host including the client may have one or more base relations stored locally.

A single test run for a two-join query takes only around 5 seconds on a 400MHz Pentium II. Each test run was comprised of the following phases:

- Reading and parsing the query from an SQL file
- Checking the canonical form of the query plan (optional)
- Analysing the cost of the canonical query plan
- Displaying canonical query plan & costs
- Optimising the canonical query plan
- Checking the optimised query plan (optional)
- Analysing the cost of the optimised query plan
- Displaying optimised query plan & costs

In this study, we are seeking to examine the behaviour of a distributed query optimiser in the context of a system with one client querying three relations each on a separate server connected by links with two alternative bandwidths: one which corresponds roughly to a 100Mbps network, and another which has been reduced to the point where it potentially passes data more slowly than it is produced or consumed. This would correspond to a real-world situation where, say, a microwave link fails, leaving only an ISDN backup carrying an
unsustainably large load. It is hoped that the use of two phase optimisation will create a query execution plan adapted to the network state, regardless of the combination of fast and slow links.

The configuration of this experiment (Figure 3 and Table 2) was designed to be large enough to allow the optimiser to suggest either left or right leaning trees while producing a manageable number of distinct cases to study.

Each of the eighteen cases is shown in Table 3. For each configuration, a series of ten optimisations was performed from different starting points in the solution space and the results collected as both a tree diagram and a table of CPU usages and network costs.

The solution space was not constrained to a particular tree form or shipping strategy: that is, the solution space spans the entire space created by bushy trees and hybrid shipping.

Transmission costs are assumed to be negligible where a producer-consumer pair resides on the same machine.

![Figure 3. Experimental Configuration](image)

**Figure 3. Experimental Configuration**

<table>
<thead>
<tr>
<th>Configuration</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>H</td>
<td>H</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>H</td>
</tr>
<tr>
<td>B</td>
<td>L</td>
<td>L</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>L</td>
</tr>
<tr>
<td>C</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>L</td>
<td>L</td>
<td>H</td>
</tr>
<tr>
<td>D</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>H</td>
<td>H</td>
<td>L</td>
</tr>
<tr>
<td>E</td>
<td>H</td>
<td>H</td>
<td>L</td>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>F</td>
<td>L</td>
<td>L</td>
<td>H</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>G</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>H</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>I</td>
<td>H</td>
<td>H</td>
<td>L</td>
<td>H</td>
<td>H</td>
<td>L</td>
</tr>
<tr>
<td>J</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>K</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>M</td>
<td>H</td>
<td>H</td>
<td>L</td>
<td>H</td>
<td>H</td>
<td>L</td>
</tr>
<tr>
<td>N</td>
<td>L</td>
<td>L</td>
<td>H</td>
<td>L</td>
<td>L</td>
<td>H</td>
</tr>
<tr>
<td>O</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>P</td>
<td>L</td>
<td>L</td>
<td>H</td>
<td>L</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>Q</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>R</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
</tbody>
</table>

**Legend:** H=High Speed (100Mbps), L= Very Low Speed

Table 2. Experimental configuration cases

4.2 Experimental Results

The following graphs are summaries of the each of the cost metrics considered for both the canonical and optimised queries. On average, there are drastic improvements in Response time of 72% (Figure 4), Completion time of 50% (Figure 5) and Network time of 48% (Figure 6), and there is a marginal improvement in the total CPU time required of 1% (Figure 7).

The improvements in response and completion time are largely due to the exploitation of parallel processing, as can be seen by comparing the total CPU time vs. the completion time for both the canonical and optimised plans. On average, the completion time marginally exceeds the total CPU time for the canonical queries, but the average completion time for an optimised query is around half of the total CPU time, indicating that parallel processing is being exploited very successfully by the optimiser. Response time and elapsed time are both relatively constant across all of the optimised plans as tuples from one base relation can be pipelined through the two joins after each of the other relations has been scanned. In general, the total CPU time is not changed significantly by optimisation and, in some cases, is actually increased. This is due to the fact that completion time is the sole optimisation metric used in this study, so plans which exploit parallelism and avoid slow network links will be preferred over those with only a lower total CPU time. In any case, the simple nature of the configuration studied means that there is little scope for varying total CPU time, since the operators used remain unchanged and any variation will be due to changes in communication overheads brought about by changed evaluation locations. This explains the increase in total CPU time in some of the cases where operators are relocated so that pages are shipped via higher bandwidth network links.

It should be borne in mind, however, that even though the cost estimates for both the canonical and optimised query execution plans do not consider any resource contention brought about by parallel processing, the effect of resource contention would be limited, since the total number of operators is small and precedence constraints prevent concurrent evaluation of some operator combinations.

Figure 6 shows average network costs for each configuration investigated. This was obtained by summing all network times (volume / bandwidth) for each link between pairs of servers across an entire tree. This provides an indication which network links are the most costly for the query evaluator to use and could form the basis of enhancements to the cost models to accurately reflect link contention brought about by concurrent transmission. As they are presented below, however, these figures only provide an indication of overall network costs. The consistently large improvement in network time indicates that the optimiser is successfully locating...
operators to route around low-bandwidth links. Note that case G corresponds to a system with a uniformly high bandwidth.

5. Conclusions & Future Work

This study demonstrates the viability of randomised optimisation techniques for determining both the order and location of query operators in a non-uniform bandwidth system, since consistent improvements in completion time and, indirectly, response time and network time were obtained by the optimiser for the configuration studied. More complex queries could be viewed as nested constructions of this configuration, so the results should hold true for these complex queries. The consideration of bandwidth limitations also extends earlier work which has confirmed the suitability of the two-phase optimisation algorithm for run-time query optimisation in distributed database systems.

An enhancement to the experiment would be to consider an additional logical location annotation which would allow an operator to be evaluated on a server which does not host any of the producer or consumer operators. This additional annotation, perhaps called floating, would allow increased parallelism and utilisation of otherwise unavailable resources with the trade-off of increased inter-server communication.

The optimiser developed in the course of this study is capable of working with query execution plans far more complex than for the simple two-join query studied in this paper and it could also be extended with different types of operators. These operators could include n-ary operators, since the transformations used will handle these as well as binary operators. Most enhancements to the model would need to be implemented only within the cost function method for each class of operator. Different caching hit-rates and varying machine speeds can also be accommodated by the implementation, offering many potential areas for investigation.

References