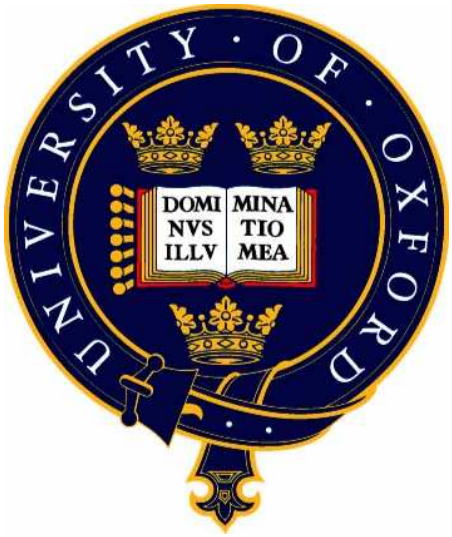
Query Optimization for Distributed Database Systems

Robert Taylor
Candidate Number : 933597
Hertford College
Supervisor: Dr. Dan Olteanu



Submitted as part of
Master of Computer Science
Computing Laboratory
University of Oxford

August 2010

Acknowledgments

I would like to thank my supervisor Dr Dan Olteanu for his incredible level of enthusiasm and encouragement throughout the project. I am also very grateful for the continuous level of feedback and organisation as well as the amount of time he has devoted to answering my queries. I feel that I now approach complex and unknown problems with enthusiasm instead of apprehension as I used to. I couldn't have had a better supervisor.

Abstract

The query optimizer is widely considered to be the most important component of a database management system. It is responsible for taking a user query and searching through the entire space of equivalent execution plans for a given user query and returning the execution plan with the lowest cost. This plan can then be passed to the *executer*, which can carry out the query. Plans can vary significantly in cost therefore it is important for the optimizer to avoid very bad plans. In this thesis we consider queries in positive relational algrebra form involving the conjunction of projections, selections and joins.

The query optimization problem faced by everyday query optimizers gets more and more complex with the ever increasing complexity of user queries. The NP-hard join ordering problem is a central problem that an optimizer must deal with in order to produce optimal plans. Fairly small queries, involving less than 10 relations, can be handled by existing algorithms such as the classic Dynamic Programming optimization algorithm. However, for complex queries or queries involving multiple execution sites in a distributed setting the optimization problem becomes much more challenging and existing optimization algorithms find it difficult to cope with the complexity.

In this thesis we present a cost model that allows inter-operator parallelism opportunities to be identified within query execution plans. This allows the response time of a query to be estimated more accurately. We merge two existing centralized optimization algorithms DPccp and IDP1 to create a practically more efficient algorithm IDP1ccp. We propose the novel Multilevel optimization algorithm framework that combines heuristics with existing centralized optimization algorithms. The distributed multilevel optimization algorithm (DistML) proposed in this paper uses the idea of distributing the *optimization* phase across multiple optimization sites in order to fully utilize the available system resources.

Contents

1 Introduction

1.1 Background and Motivation

The query optimizer is widely considered to be the most important part of a database system. The main aim of the optimizer is to take a user query and to provide a detailed plan called a Query Execution Plan (QEP) that indicates to the executer exactly how the query should be executed. The problem that the optimizer faces is that for a given user query there exists a large space of different equivalent QEPs that each have a corresponding execution cost. The plans are equivalent in the sense that they return the same result for the user query but the cost of plans may differ by orders of magnitude. In a centralized database system an estimate of the number of I/Os performed is typically used as a cost metric for a plan. If the optimizer choses a plan with a poor cost the execution can take several days while another plan may exist that performs the execution in seconds [5].

The optimization problem encompasses the join ordering problem, which has been proven to be NP-complete [17]. This means that no polynomial time algorithm currently exists that can find the optimal plan for all sizes of queries in a feasible time frame. One of the most widely used optimization algorithms in commercial databases, such as IBM DB2 [11], is based on Dynamic Programming (DP) and can be seen in Figure 12. However, DP has a worst case time complexity of $\mathcal{O}(3^n)$ and space complexity of $\mathcal{O}(2^n)$ where n is the number of relations involved in the query. For complex queries* DP would exhaust the available memory and would take an unfeasible amount of time to return the optimal plan. Unfortunately, complex queries are being encountered more frequently to meet the increasingly complex requirements of the users/companies of today.

Companies with offices in different cities find that they must provide access to the same data from different locations, which leads them to making use of *distributed* database systems. Optimizers in these systems face the highly complex problem of determining the best execution plan in the presence of multiple possible relation copies located at different sites. The execution site for each operator has to be determined and the cost of transferring relations between sites has to be included in the cost model in addition to the I/Os performed. The optimizer must also be able to recognise the opportunities for parallelism that arise as a result of the distributed setting and adjust its cost model to recognise plans that utilize these possible concurrent execution opportunities. For example, it may be possible to utilize two different sites in a distributed system to execute different parts of a

*By the term 'complex queries' we mean queries that involve a large number (≥ 50) of relations.

given query plan concurrently thereby reducing the total execution time. A fairly simple alteration of the classic DP algorithm was suggested in [9] that enabled it to be applicable in the distributed database setting. However, the worst case running time in this case is $\mathcal{O}(s^3 \cdot 3^n)$ where s is the number of execution sites in the system. For even moderately sized systems using this algorithm to optimize even fairly simple queries becomes unfeasible.

The main aim of this thesis is to produce a query optimizer that is capable of optimizing large queries (involving ≥ 50 relations) in a distributed setting. We assume that the distributed setting is homogeneous in the sense that all sites in the system run the same database management system software [16]. We consider *complete* relations to possibly be replicated at different sites in the system. Extending this assumption to include fragmented relations is left as further work. We wish to produce an optimizer that is capable of recognising opportunities for *inter*-operator parallelism within QEPs. Given a query the optimizer should aim to produce the plan with the shortest execution time or *response time*.

1.2 Related Work

Several approaches have been proposed to enumerate over the search space of equivalent plans for a given user query. These include randomized solutions such as Iterative Improvement (II) and Simulated Annealing (SA) [23] and heuristic-based methods such as the minimum selectivity heuristic [19]. This heuristic is described further in section 2.3.2. Query re-writing [25] and query simplification [12] techniques have also been recently explored where the query graph is restricted in an attempt to reduce the complexity of the optimization.

Iterative Dynamic Programming (IDP) was proposed in [9] in 2000 to overcome the space complexity problem of DP. However, DP and IDP do not consider the query structure or *join graph*, which consequently leads them to consider cross products. Cross products are costly and result from joining relations together that do not have a join condition present. The DPccp algorithm proposed in [21] in 2006 uses the join graph of a query to perform dynamic programming without considering cross products. DPccp still has the same worst case running time as DP (in the case where a clique query is being optimized) but in practise it produces considerable savings in time when optimzing chain and cycle queries.

1.3 Outline and Contributions

The contributions of this thesis are as follows.

- We tailor cost models suited for parallel database systems from [7] to be applicable in the *distributed* database setting. This involves taking concepts such as identifying tasks and scheduling and adapting them to be suitable in our distributed setting. We give a focus on modelling inter-operator parallelism, while cost models for parallel database systems typically focus on exploiting *intra*-operator parallelism, i.e. using multiple execution sites to execute a single operator concurrently.
- We combine two existing centralized optimization algorithms DPccp [21] and IDP1 [9] to produce an improved version of IDP1, which we name IDP1ccp. It is clear from the experimental results of [9] that this algorithm was considered and indeed some results were produced. However, DPccp was produced in 2006 while IDP1 was produced in 2000. In [9] only small examples of a single chain and a single star query were considered, which leads me to believe that the authors used a degree of code hardwiring and did not produce a general algorithm capable of operating over arbitrary join graphs as IDP1ccp is.
- We propose the novel multilevel (ML) optimization algorithms with the intention of being able to optimize larger queries and queries in larger distributed settings than IDP1ccp is able to due to limitations on its time complexity.
- The project Java code and experimental data can be found at the following URL.

http://www.comlab.ox.ac.uk/people/dan.olteanu/projects/distributedoptimization.html

This thesis is organized as follows.

- Section 2 introduces background information on distributed database systems, distributed query processing and query optimization in a centralized database system.
- Section 3 highlights the challenges present when implementing a query optimizer in distributed database systems.
- Section 4 presents our query execution plan cost model that is capable of modelling concurrent execution between multiple operators, i.e. *inter*-operator parallelism.

- In section 5 we present two existing centralized query optimization algorithms DPccp and IDP1 and combine them to produce a third algorithm, which we name ICP1ccp. This algorithm is then used as a benchmark in experiments involving our multilevel optimization algorithms.
- Section 6 presents the novel multilevel (ML) optimization algorithms that have been designed to operate in the event that IDP1ccp cannot operate due to the time complexity of the algorithm. We propose a sequential ML algorithm (SeqML) and then a distributed ML algorithm (DistML) that uses multiple *optimization sites* with the aim of better using the available system resources for query optimization (and not just query execution).
- Section 7 gives implementation details of the optimization algorithms presented in this thesis. Here we discuss internal structures and file formats used for query execution plans, join graphs, the system catalog as well as giving class diagrams showing the design of the query optimizer.
- In section 8 the experimental results are shown that indicate the comparative performance of SeqML, DistML and IDP1ccp with regards to plan optimality.
- Appendix A gives variations of existing algorithms required by the IDP1ccp algorithm. These variations are small but are important to the algorithm.
- Appendix B gives an example of the file format of a system catalog and a randomly generated query.
- Appendix C gives a proof of the number of connected components for a chain query in terms of the number of relations involved in the query.
- Appendix D gives the design of the query generator used to produce queries for the experiments.
- Appendix E presents experimental results comparing the ML algorithms with IDP1ccp in the distributed setting containing one execution site and nine execution sites.
- Appendix F gives the maximum k values that the ML algorithms and IDP1ccp can take in the experiments (due the existence of an optimization time cap).
- Appendix G gives experimental results for determining the best k parameter for the ML algorithms in the distributed setting containing one execution site and nine execution sites. For brevity only the results for 100-Query are shown.

2 Preliminaries

In this section we present some background information regarding distributed database systems including desirable properties, architectures and query execution techniques. We then discuss some of the conceptual problems encountered when designing a query optimizer for a centralized database system and we present some of the existing solutions.

2.1 Distributed Database Systems

A distributed relational database is a distributed database consisting of multiple physical locations or *sites* and a number of relations. Relations may be replicated and/or fragmented at different sites in the system. The placement of data in the system is determined by factors such as local ownership and availability concerns. A distributed database management system should not be confused with a *parallel* database management system, which is a system where the distribution of data is determined entirely by performance considerations. When every site in the system runs the same DBMS software the system is called homogenous. Otherwise, the system is called a heterogenous system or a multidatabase system [16].

Having data distributed across multiple sites is particularly advantageous for large organizations who have offices and their work force distributed across the world. As well as offering access to common data from multiple locations, having distributed data can offer other benefits including improving the availability of data. For example, by distributing data we avoid the single point of failure scenario where data resides at a single site that can fail. By replicating data across several sites in the event of a site failure there would be more chance of having an alternative copy of the required relation available. This would allow the service requesting the relation to continue without being disrupted. Replicating data not only improves availability but also improves *load distribution*. By creating copies of relations that are in high demand we reduce the load on owning sites, which may have an impact on reducing the number of site failures. By replicating relations to sites where they are most commonly used network accesses can be reduced, which can improve the performance for users at those sites.

Two desirable properties that a distributed database management system should possess are Distributed Data Independence and Distributed Transaction Atomicity. Distributed Data Independence refers to the fact that a user should be oblivious to the location of data that they query or manipulate. This also implies that query op-

timization in a distributed system should be systematic as in a centralized system in the sense that the optimzer should use a cost model (incorporating local disk I/O and network costs) to determine where to fetch data from, where to execute evaluation operators and in what order. Distributed transaction atomicity refers to the fact that users should be able to perform transactions in a distributed environment and these transactions should be atomic. This means that if a user transaction completes then the results should be visible to all other users of the system and in the event that the transaction aborts the system should be presented to all users in the state that existed before the transaction was initiated [16].

2.1.1 Distributed Database System Architecture

Distributed database systems traditionally adopt a *client-server* architecture consisting of client and server processes. In a client-server architecture there exists a clean seperation of responsibility between client processes and server processes. Data is stored at server sites that run server processes that provide access to their local data to client processes. Client sites run client processes that provide an interface between users and the data residing in the system.

In a *collaborating server* system we have much the same system configuration as we have in a peer-to-peer (p2p) network, where each process or *node* in the system can act as both a client and server. In this case data is stored at all nodes and it is possible for a node to receive a transaction request that involves data that is not stored locally to it. If a node receives a transaction request requiring access to data stored at other sites the node will collaborate with the appropriate sites until the transaction is fulfilled or is aborted. An example p2p system consisting of four sites (s_1, s_2, s_3 and s_4) is shown in Figure 1. The system contains the three relations R_1, R_2 and R_3, some of which are replicated at multiple nodes. In the event where a transaction request is issued at s_1 that involves relations other than R_1, s_1 will have to collaborate with other sites in the system in order to carry out the transaction [16].

2.2 Distributed Query Processing

2.2.1 Catalog

In a centralized database system the catalog is used to primarily store the schema, which contains information regarding relations, indices and views. The meta-data stored for relations includes the relation name, attribute names and data types and

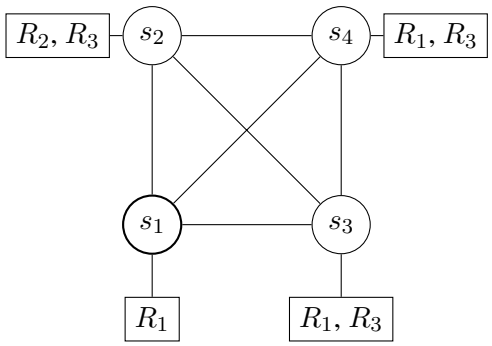


Figure 1: p2p network

integrity constraints. Various statistics are also stored in the catalog such as the number of distinct keys in a given attribute and the cardinality of a particular relation. These statistics aid the query optimizer in estimating the size of intermediate results of execution plans, which allows the optimizer to determine an estimate of the cost of plans. Information about the current system state is also made available in the catalog, including the number of buffer pages in the buffer pool and the system page size in bytes [16].

In a distributed DBMS the catalog has to store additional information including the location of relations and their replicas. The catalog must also include system wide information such as the number of sites in the system along with their identifiers. An issue that arises in a distributed setting is where to place the catalog in the system. Two alternatives include storing the catalog at a single site or replicating it at all sites. Storing the catalog at a single site introduces a single point of failure in the system. If the site containing the catalog fails then the entire system cannot access the catalog, which may cause the entire system to come to a halt. Placing a copy of the catalog at a single site causes the site to become burdened by many catalog queries from other sites, which may lead to performance degradation. Creating replicas of the catalog at each site eliminates the single point of failure problem and also reduces the network communication required as all sites do not have to access the catalog remotely. However, in a system where catalog updates are frequent, much time may be invested in synchronising the catalog replicas, which may degrade system performance [8].

2.2.2 Query Evaluation Techniques

When considering the evaluation of query operators in a distributed setting we need to consider not only the I/O cost on local disk but also the network costs, which have a significant contribution to the overall query execution cost. Any cost model used in a distributed setting must also take into account the site that issues the query S_q so that we include the cost of shipping the result from the site at which it is materialized to S_q. In the following discussion we consider the case where relations are stored completely at sites and are not fragmented, as this is the approach taken in this thesis. In this case performing selections and projections is straight forward as we simply use centralized implementations of these algorithm on a relation at a particular site and then ship the results to S_q if necessary [16].

When considering the join between two relations R_1 and R_2 in a distributed setting there are a number of situations that must be taken into consideration when determining the most efficient means of carrying out the join. Situations involving possible shipping of complete relations between sites are discussed below. Techniques that aim to reduce the network useage when performing cross site joins include the use of **Semi-joins** and **Bloomjoins**. These techniques were not considered in this thesis and are left for further work. Details of these techniques can be found in [16] for the interested reader.

- If both relations are present at the same site the join can be performed entirely at the local site. As communication costs are more expensive than disk costs in general it is likely that this approach will result in a plan with the least cost.
- If both relations R_1 and R_2 reside at different sites, S_1 and S_2 respectively, the join can be performed at either site. If one of these sites S_1 happens to be the site at which the query was issued, henceforth known as the *query site*, it is likely that the optimal query plan will involve shipping R_2 from S_2 to S_1 and performing the join at S_1. However, consider the situation where R_2 is very large, R_1 is very small and the result of joining R_1 and R_2 is very small. In this case the optimal plan may involve shipping R_1 to S_2 and performing the join at S_2. The small result would then be returned to the query site.
- It is also possible to choose a join site S that does not contain either of the relations. This would involve shipping both relations to S. In the event where S is the query site the cost of shipping the result to the query site is saved. This is particularly useful in the event where the result of a join is much larger than the size of the input relations. In the event where S is not

the query site this approach may have little benefit and most likely will result in a plan with a poor cost. However, when a join between two relations is part of a larger query this approach can be benefical, as discussed in Section 4.

When a relation R is shipped between different sites the tuples in R are sent across the network using a network protocol such as UDP or TCP [6]. A network protocol is capable of sending a maximum data payload size with each unit of communication. For example, when using UDP with IPv4 the unit of communication is the UDP datagram, which has a maximum data payload size of 65,507 bytes. Assume that the size of tuples in R are much less than 65,507 bytes. If we transmit R a tuple at a time the maximum size of a datagram is not utilized fully. This causes an unneccessarily large number of messages to be sent. Instead of sending tuples one at a time tuples can be sent in blocks. This technique is called *row blocking* and is used by almost all distributed database systems [8]. As well as fully utilizing the underlying network protocol transmission unit row blocking has an additional advantage that it contributes to maintaining a smooth rate of execution of queries. If a site is using received tuples as the pipelined input to a join the rate of processing will be determined by the rate at which tuples are received. If the block size is small the receiver may perform the join using tuples from a received message and then have to wait for further tuples to arrive in order to continue evaluating the join. This causes idle periods at the execution site. Ideally the join would be continuously supplied with input tuples until the join operation was completed. By having larger block sizes the receiver can buffer tuples that can be processed while waiting to receive further messages.

2.3 Query Optimization in a Centralized Database system

When a user issues a query in a centralized DBMS the optimizer must produce a detailed Query Execution Plan (QEP) that can be passed to the executer. The executer can then follow this plan to carry out the evaluation of the query and return the results to the user. A typical SQL user query is shown in Figure 2. In this query the user is specifying that they wish to receive the price of all items ordered by the customer named 'Joe'. The relations involved in the query are named in the FROM clause as Cust, Item and Ord that store information about customers, items and orders respectively. The output attributes are specified in the SELECT clause and the join conditions are specified in the WHERE clause. Join conditions are predicates that take tuples t from the cartesian product of two input relations and maps them into the boolean set $S = \{true, false\}$, where the elements of S

indicate whether or not t will be present in the output [19]. For example, if we have a join $R_1 \bowtie_{a=b} R_2$, only tuples from $R_1 \times R_2$ where $R_1.a = R_2.b$ will be present in the output. Here, $R_1.a$ denotes the attribute named a from relation R_1. In this thesis we consider only the equi-join operation, that is only join conditions that contain an equality between attributes.

```
SELECT price FROM Cust AS C, Item AS I, Ord AS O
WHERE C.ckey1 = O.ckey2
  AND O.okey2 = I.okey3
  AND C.name  = 'Joe'
```

Figure 2: SQL Query

An optimizer can combine *selectivity* information from the system catalog with an SQL query in order to produce a *join graph* representation of the given query. The selectivity σ of a join $R_1 \bowtie_{a=b} R_2$ is defined as the ratio of the number of result tuples $nt_{1,2}$ to the cardinality of the cartesian product of R_1 and R_2 i.e.

$$\sigma(R_1 \bowtie_{a=b} R_2) = \frac{nt_{1,2}}{|R_1 \times R_2|}.$$

A join graph consists of a number of vertices, representing relations, and edges, representing join conditions. An edge between vertices v_1 (representing relation R_1) and v_2 (representing R_2) indicates that a join condition exists between R_1 and R_2. Edges are annotated with the join condition and associated selectivity. An example join graph constructed from the query above can be seen in Figure 3. In this example, the selectivity for $Cust \bowtie_{ckey1=ckey2} Ord$ obtained using the system catalog is 0.01. Four common structures of join graphs are chain, cycle, star and clique, which can be seen in Figures 4-7.

For each user query there can exist a large number of equivalent plans that produce the same result but which have different execution costs. The cost of a plan is estimated using a *cost model* that is based upon examining the number of times the disk is accessed during execution of the plan. The problem that the optimizer

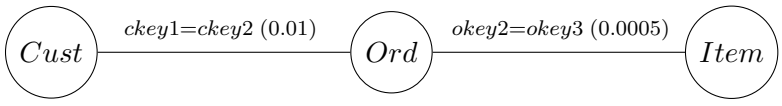


Figure 3: Join Graph

Figure 4: Chain Query

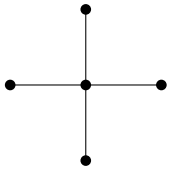


Figure 5: Star Query

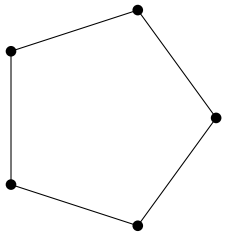


Figure 6: Cycle Query

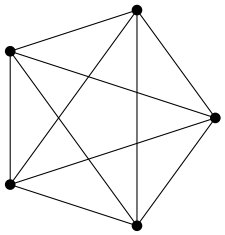


Figure 7: Clique Query

faces is that equivalent plans can have costs that vary by orders of magnitude so choosing the wrong plan can result in a user having to wait days for a result instead of seconds [5]. Ideally the query optimizer would use an *enumeration algorithm* to enumerate over the entire *search space* and find the plan with the lowest cost. However, in practise the expectation of an effective optimizer is often just to avoid very poor plans.

2.3.1 Search Space

The search space considered during query optimization consists of the set of equivalent plans that produce the result to a given user query. QEPs take the form of a binary tree consisting of relational operator nodes. The leaf nodes represent an access method to a relation while inner nodes represent query evaluation operators such as selections (σ), projections (π) and joins ($\bowtie$). The output of these operators can either be materialized or pipelined. When an operator is annotated as *materialized* this indicates that the result of the operator will be written to disk as a temporary relation. However, if a relation is annotated as *pipelined* this means that each tuple from the operator is maintained in memory and passed to the input of parent operator as they become available. The flow of data in a plan is from the leaves up to the final operation at the root. A plan is said to be *left deep* if the inner child of a join node is always a base relation, as depicted in Figure 8. Similarly a plan in which the outer child of every join node is a base relation is referred to as

right deep. Otherwise a plan is known as *bushy*.

Typically QEPs are constructed in two phases. The initial stage involves creating the basic *algebraic* binary tree structure, which imposes an ordering on the base relations and the operations on the base relations. The second stage then adds more detail to the algebraic tree by specifying *physical* implementation details such as join algorithms, e.g Sort Merge Join (SMJ), and intermediate result output methods (materialized or pipelined). An example of an algebraic plan for the evaluation of the SQL query in Figure 2 is shown in Figure 8 and an example of a completed plan with physical annotations can be seen in Figure 9. A query executer can follow a QEP by performing the operations specified as a result of performing a post order traversal on the tree.

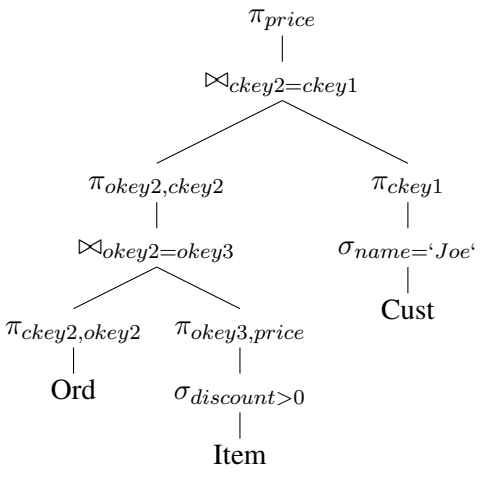


Figure 8: Algebraic Plan

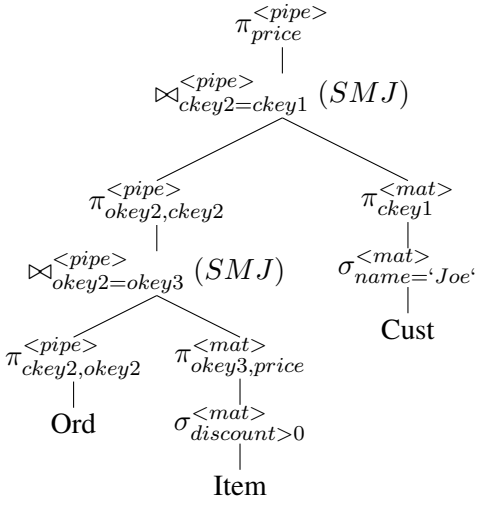


Figure 9: Physical Plan

Let us consider the size of the search space given a basic user query Q involving joining the relations R_1, R_2, ..., R_{n+1}. n join operations are required to join $n+1$ relations implying that query execution plans for Q will take the form of a binary tree with n inner join nodes and $n+1$ leaf nodes corresponding to the relations. The commutativity property of the join operation results in the search space consisting of all structures of binary trees and for each tree structure we can permute the relation leaf nodes. The number of unique binary trees containing $n+1$ leaf nodes is $\frac{(2n)!}{(n+1)!n!}$ [10] and the number of permutations of relations for each tree structure is $(n+1)!$. The resulting size of the algebraic search space is therefore $\frac{(2n)!}{n!}$. Figure 10 shows the possible binary tree structures for a query involving 3 relations and Figure 11 shows the possible permutations produced for a single tree structure for

the relations A, B and C.

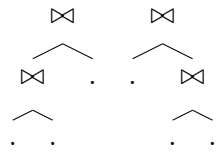


Figure 10: Binary tree structures

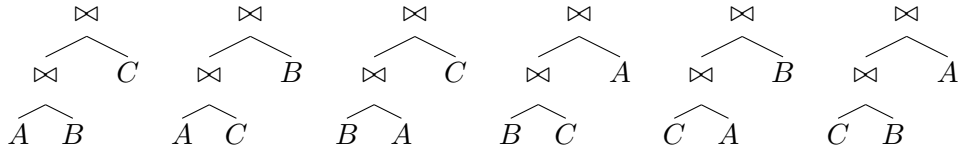


Figure 11: Permutations of leaves for a single tree structure

The size of the algebraic space acts as a lower bound to the space of physical plans, where the choice of implementation algorithms for operators causes the space to increase. For a query involving 10 relations the search space contains at least 17,643,225,600 plans [13], which may be too large to enumerate in a brute force manner in a feasible time frame.

2.3.2 Enumeration Algorithms

Enumeration algorithms typically fall into one of three catergories - exhaustive search, heuristic-based or randomized.

- **Exhaustive Search** algorithms have exponential worst case running time and exponential space complexity, which can lead to an algorithm requiring an infeasible amount of time to optimize large user queries, i.e. queries involving more than about 10 relations. Since exhaustive algorithms enumerate over the entire search space the algorithm will always find the optimal plan based upon the given cost model. However, the problem of finding the best plan is NP-complete, which implies that a polynomial time algorithm to find the best plan for queries of all sizes is unlikely to be found [9].

  The traditional dynamic programming (DP) enumeration algorithm, shown in Figure 12, is a popular exhaustive search algorithm, which has been used in a number of commercial database management systems including IBM DB2 [11]. As with other exhaustive search algorithms DP has an exponential worst case running time and space complexity of $O(3^n)$ and $O(2^n)$

respectively [14, 22]. Let $C(p)$ denote the cost of a subplan rooted at the QEP node p. The principle of optimality states that if two plans differ only by a single subplan rooted at p then the plan with the lowest $C(p)$ will also have a lower complete plan cost [4]. The DP algorithm uses the principle of optimality to build the optimal plan by considering joining optimal subplans in a bottom up fashion. The algorithm initially finds the best access method for each relation (lines 1-5). All possible plans consisting of 2 relations are then built by using the best access methods as building blocks. The function $joinPlans$ used in line 11 creates a new join node with its parameters as children. These plans are then pruned, using the $prunePlans$ function, to leave the plan with the lowest cost for each combination of 2-relation plans. All possible 3-relation plans are then constructed by considering joining optimal 2-relation plans with the optimal access plans. In general when building optimal subplans involving k relations the building block join pairs that are used are the optimal subplans of size i and the optimal subplans of size $k-i$ for all $0 < i < k$ (lines 6-15). For a query involving n relations the best plan is built after n passes [13].

The $finalizePlans$ function uses heuristics to place other relational operators other than joins (e.g. selections and projections) along with materialized/pipelined annotations onto the obtained plans. In some instances the $prunePlans$ function cannot prune all plans for a given relation combination as the plans are said to be 'incomparable'. Incomparable plans arise as a result of plans that possess an *interesting order* [18] i.e. plans that are sorted in some manner. Even though an interesting order plan p may have a higher execution cost than another plan q, for a particular relation combination, the sorted property of p may reduce the execution time of a future join operation. This implies that we cannot discard plans with interesting orders even if they do not have the lowest plan cost. Plans involving all query relations are comparable as there are no future join operations to add to the plan hence the final prune can be performed disregarding interesting orders to obtain the single best plan (line 17-18).

- **Heuristic-based** algorithms were proposed with the intention of addressing the exponential running time problem of exhaustive enumeration algorithms. Heuristic-based algorithms follow a particular heuristic or rule in order to guide the search into a subset of the entire search space. Typically these algorithms have polynomial worst case running time and space complexity but the quality of the plans obtained can be orders of magnitude worse than the best possible plan.

Algorithm DYNAMIC-PROGRAMMING($R = \{R_1, R_2, ..., R_n\}$)

Input: The set of relations ($R = \{R_1, R_2, ..., R_n\}$ involved in the query.
Purpose: To build the optimal plan from optimal subplans using Dynamic Programming.
Output: The optimal query execution plan.

```
1  ▷ Produce optimal plans consisting of 1 relation.
2  for i ← 1 to n do {
3     opt-plan({R_i}) ← ACCESS-PLANS({R_i})
4     PRUNE-PLANS(opt-plan({R_i}))
5  }
6  for i ← 2 to n do {
7     ▷ Produce optimal subplans consisting of i relations.
8     for all S ⊂ R such that |S| = i do {
9           opt-plan(S) ← ∅
10          for all O ⊂ S where O ≠ ∅ do {
11             opt-plan(S) ← opt-plan(S)∪
                                JOIN-PLANS(opt-plan(O), opt-plan(S \O))
12             PRUNE-PLANS(opt-plan(S))
13          }
14    }
15 }
16 FINALIZE-PLANS(opt-plan(R))
17 PRUNE-PLANS(opt-plan(R))
18 return opt-plan(R)
```

Figure 12: Traditional Dynamic Programming Enumeration Algorithm

The *Minimum Selectivity* algorithm is an example of a heurstic-based algorithm. In Minimum Selectivity we create a skeleton left deep plan T with void gaps where the relation leaves should be located. Let the relations placed into this skeleton plan be denoted as R_{used} and the remaining relations involved in the query be denoted as R_{rem}. Initially R_{rem} contains all relations involved in the query. The relation R_1 with the smallest cardinality is chosen first and placed into the lower left hand leaf of the tree. When a relation is added into T it is removed from the R_{rem} set and added into the R_{used} set. Let I denote the intermediate result of evaluating the largest complete subtree of T. While R_{rem} is non empty we continue to choose a relation $R_i \in R_{rem}$ such that $\sigma(I \bowtie R_i) = min\{\sigma(I \bowtie R_j)|R_j \in R_{rem}\}$ and add it to the lowest available relation position in the tree[19]. The mo-

tivation behind using this heuristic is that most good plans have small intermediate results. A number of other heuristics that have been proposed include greedy heuristics [15, 20] and the Iterative Dynamic Programming (IDP) variants [9], which are discussed further in section 5.

- **Randomized** algorithms consider the search space as a set of points each of which correspond to a unique QEP. A set of *moves* M is defined as a means to transform one point in space into another i.e. a move allows the algorithm to jump from a point in space to another. If a point p can be reached from a point q using a move $m \in M$ then we say that an edge exists between p and q. A number of randomized algorithms exist such as Iterative Improvement (II) [24, 1], Simulated Annealing (SA) [23, 1], 2-Phase Optimization(2PO)[24] and genetic algorithms [2].

  In II we select a randomly chosen starting point in space. Neighbours are then randomly chosen until a solution point is found that has a lower cost than the current point. We move to this new point and repeat the process. Once a local minimum is found we determine if this local minimum has a lower cost than any previously found local minima. If so, we record the point along with its cost. We then start the process again by choosing a new initial starting point to explore from. After a set time the local minimum with the lowest cost is returned. This algorithm can return plans with much higher costs than the global optimal solution in situations where the solution space contains many local minima. SA addresses this problem by allowing moves that increase the cost of a plan. These moves are allowed with a given probability. This reduces the chance of the algorithm becoming trapped in local minima whose cost is much greater than that of the global minima. II is very good at exploring a large part of the solution space and finding local minima quickly whereas SA is very good at covering a smaller neighbourhood of the solution space in greater detail. 2PO combines II and SA by first applying II to discover many local minima. The local minimum with the lowest cost is then chosen and SA is applied with this initial point.

2.3.3 Cost Model

In order to allow an optimizer to be able to find an 'optimal' plan it is essential that the optimizer incorporates the use of a cost model that accurately estimates system resources used for individual operators and complete plans. In a centralized DBMS cost models are based upon determining the number of pages that are read from or written to disk. We make the assumption that system memory is not large enough

to store complete relations therefore the dominant operations that will contribute to the execution time of a plan are disk I/O events. In this thesis we consider SPJ queries, i.e. queries involving selections, projections and join operators. Below we discuss the cost of the selection, projection and Sort-Merge Join operators.

Selection operators take the form $\sigma_p(R)$ where R is a relation and p is a predicate that indicates whether or not a particular tuple of R should be included in the result. In the event where no index exists on relation R we have to perform a complete scan of R and determine whether each tuple satisfies the selection predicate. The cost of performing a selection in the absence of indices is therefore the cost to read relation R from disk plus the cost to write the result to disk. If R consists of M pages then the cost to read R is simply M I/Os. In order to determine the output cardinality the selectivity of the selection would have to be determined. The selectivity of the selection operator can be established by consulting with the system catalog. The catalog stores statistics such as the size of relations present in the system and the number of distinct values that occur in a given field of a particular relations. The catalog can then provide estimates on the result sizes of operators by using these statistics. For example, consider the equality selection $\sigma_{a=x}(R)$, which requires the tuples in R whose a attribute value equals x. By assuming a uniform distributed of values in the selection attribute the size of the result can be estimated to be $\frac{|R|}{d_a}$ where d_a is the number of distinct values occuring in a and $|R|$ is the total number of tuples present in R [16].

Projection operators take the form $\pi_l(R)$ where l is a subset of attributes belonging to the schema of R. By default when issuing SQL queries any duplicates that occur in the output as a result of performing a projection are not eliminated. When duplicates are not eliminated the cost of performing a projection is equal to the cost of scanning R plus the cost of writing out the result. As with selections the cost of scanning R is simply M I/Os. Let the size of a tuple in R be b_R bytes (assuming fixed size tuples) and the size of a tuple containing only the attributes l to be b_l bytes. The number of output pages can then be estimated by the following.

$$C_{\pi_l(R)} = \lceil M \times \frac{b_l}{b_R} \rceil$$

In total the number of I/Os required to perform a projection without duplicate elimination is $M + C_{\pi_l(R)}$.

The Sort-Merge join algorithm comprises of two phases - the sorting phase and the merging phase. Let us consider the case where the Sort-Merge join algorithm

is being used to evaluate $R_1 \bowtie_{a=b} R_2$, where R_1 consists of M pages and R_2 consists of N pages. In the sorting phase each relation is sorted on their join attribute. In this case R_1 would be sorted on attribute a to produce the sorted relation S_1 and similarly R_2 would be sorted on attribute b to produce the relation S_2. Unnecessary sorting can be avoided if it is recognised that a relation or both relations are already sorted on their join attribute. Assuming that external merge sort is the sorting algorithm used the sorting phase involves $MlogM + NlogN$ I/Os.

As a result of the sorting phase tuples will be grouped together into partitions in each relation S_1 and S_2 by common join attributes values. This means that if we have a partition $p \subseteq S_1$ all tuples $t \in p$ will have a common a value. We can avoid enumerating all tuples in the cross product by identifying qualifying partitions pairs (p, q) where $p \subseteq S_1$ and $q \subseteq S_2$ such that for all $t \in p$ and all $s \in q$ $t.a = s.b$ where $t.a$ denotes the value of attribute a in tuple t. The merging phase involves scanning through S_1 and S_2 looking for qualifying partitions pairs (p, q) and then outputting the result of $p \times q$ to the output file. If we assume that S_2 partitions are scanned at most once or partition pages are stored in the buffer pool the merging phase requires $N + M$ I/O accesses [16]. The total number of I/O accesses using Sort-Merge join (without considering the cost to write out the result) is therefore given as follows.

$$C_{SMJ}(R_1 \bowtie R_2) = MlogM + NlogN + M + N$$

The number of I/Os required to write the output of a join to disk can be estimated using the following formula [19].

$$C_{output}(R_1 \bowtie R_2) = \frac{\sigma_{R_{1,2}} \times |R_1| \times |R_2| \times t_{R_{1,2}}}{p_b}$$

Here, $\sigma_{R_{1,2}}$ denotes the selectivity of the join between R_1 and R_2, $t_{R_{1,2}}$ denotes the resulting tuple size in bytes and p_b denotes the size of a page in bytes. The total cost of SMJ, including materializing the result, is therefore given by $C_{SMJ}(R_1 \bowtie R_2) + C_{output}(R_1 \bowtie R_2)$. The Sort-merge join algorithm is given in [16].

The cost of an entire plan can be obtained by summing the cost of individual operators encountered as a result of performing a post-order traversal of the plan.

3 Query Optimization Challenges in a Distributed Environment

In this section we highlight the main conceptual problems encountered when designing an optimizer for a homogenous distributed database. The first problem relates to the size of the search space. We saw in section 2.3.1 that the size of the search space in a centralized system becomes huge with even moderately sized queries[†]. In a distributed database the existance of relation copies at multiple sites and the number of execution sites contribute to making the search space even larger. The second problem emphasises the necessity of identifying inter-operator parallelism opportunities within a given plan in order to be able to determine the minimum possible total execution time of the plan accurately.

3.1 Search Space Cardinality

The problem of finding the optimal query execution plan in a centralized database system is an NP hard problem. The size of the physical plan search space for a query involving $n+1$ relations in the centralized case has a lower bound of $\frac{(2n)!}{n!}$, as this is size of the algebraic plan space (section 2.3.1). In a distributed setting the search space will be considerably larger as we have the opportunity to execute joins at any site and use any available copies of relations residing at different sites. We now derive a lower bound on the algebraic search space cardinality $|S|$ of a query involving $n+1$ relations in the worst case in the distributed setting. The worse case consists of all system relations being present at all sites. We proceed by considering three levels on annotations on plans. The first level consists of plans with join and relation nodes without any site annotations. An example of plans in this level can be seen in Figure 11. The next level of plans contain site annotations at leaf nodes only (e.g as in Figure 13) and the final level consists of complete plans with all node site annotations (e.g as in Figure 14). In Figure 14 $\bigcirc_{Si \rightarrow Sj}$ denotes a ship node where data is shipped from Si to Sj.

The number of unique plans with $n+1$ relation leaves is given by $\frac{(2n)!}{n!}$ (section 2.3.1). This corresponds to the number of plans in level 1. In a distributed setting we have the additional opportunity of using a relation from any resident site. Let s be the number of sites present in the system. Then for each tree structure in level 1 it is possible to obtain s^{n+1} tree instances in level 2 by considering all combinations of relation resident sites. An example of level 2 plans can be seen in

[†]queries that involve more about 10-15 relations

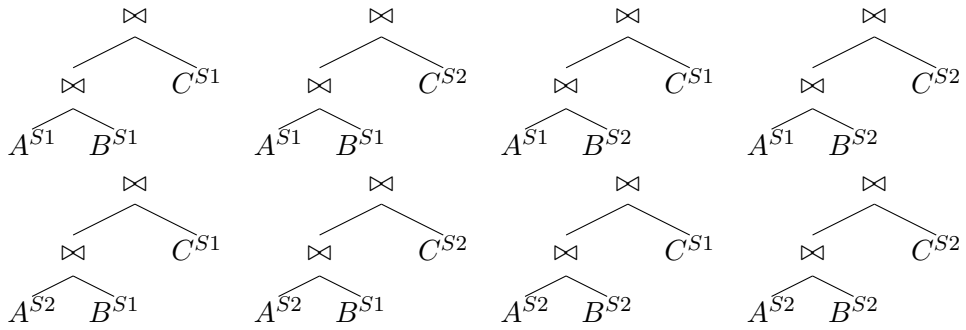


Figure 13: Combination of leaf sites for a single permutation tree structure

Figure 13, where the system contains 2 sites $S1$ and $S2$. For each tree instance in level 2 we can then consider the locations at which the joins are carried out. By performing all combinations of join locations of the n joins for each tree instance in level 2 we obtain s^n unique level 3 plans. The total number of level 3 plans correspond to the lower bound on the number of plans in a distributed setting and is given by the following formula.

$$|S| \geq \frac{(2n)!}{n!} s^{n+1} s^n = \frac{(2n)!}{n!} s^{2n+1}$$

If we consider the case where we wish to optimize a query involving 10 relations in a small system containing only two sites the number of plans in the search space will be at least 3.7×10^{16}!

3.2 Establishing Query Execution Plan Cost

In a centralized DBMS the cost of a plan is determined by aggregating individual operator I/O estimations. Using the number of I/Os as a cost metric is a good indication of the total execution time or *response time* of the plan. It is important to have a cost model that closely reflects the actual execution cost of plans otherwise it may be the case that good plans are dismissed erroneously by the optimizer while enumerating through the search space. When considering the cost model in a distributed DBMS it is not enough just to consider the I/O cost. The cost of sending relations between sites via the network has a significant contribution to the overall execution time and should also be included in the cost model. The network transmission rate would therefore have to be available to the optimizer. However,

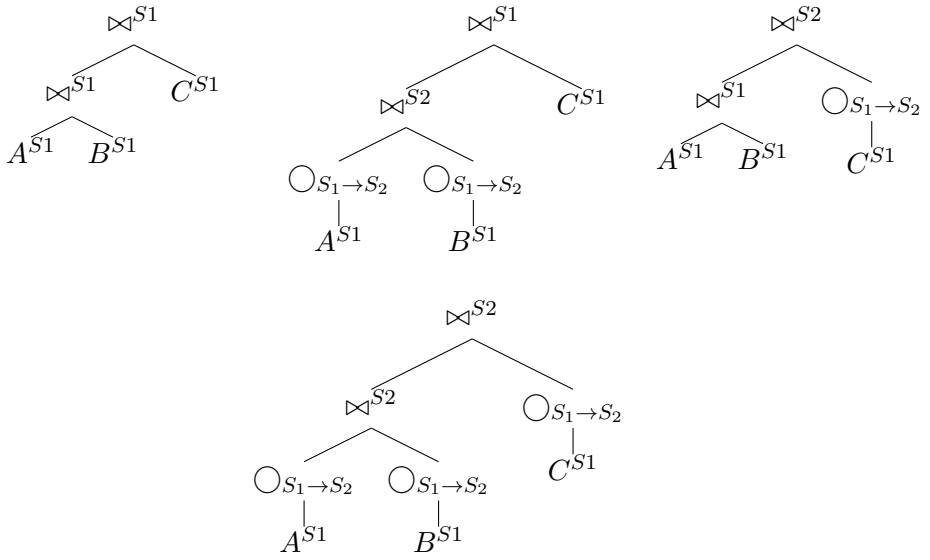


Figure 14: Combination of join sites for a single permutation tree structure

the network transmission rate is dependent on the network traffic and therefore can fluctuate over time. It is possible to make an optimistic assumption that the network traffic is low and use the baseline transmission rate of the network. However, in cases where network traffic varies significantly the optimizer would have to have a means of monitoring network traffic in order to determine the network transmission rate at runtime.

In a distributed DBMS there exists the possibility of executing parts of an execution plan in parallel. There are two main *sources* of parallelism - intra-operator and inter-operator. Intra-operator parallelism refers to the situation where a *single* operator is executed by multiple processors concurrently. This is only possible in situations where relations are fragmented across sites. Inter-operator parallelism is the term given to the case where *two* operators are executed concurrently. This includes the data-dependent case where two operators adopt a pipelining producer-consumer relationship and also the data-independent case where two operators independently execute concurrently. As well as sources of parallelism there are also *deterrents* of parallelism, which include resource contention and data dependencies. Resource contention refers to the situation where multiple processes wish to access the same resource. Data dependencies exist when an operator is required to wait for its input operators to complete before it is able to proceed [4]. As mentioned previously, in a centralized setting the response time of a given execution plan can be estimated by considering resource consumption i.e disk useage. How-

ever, in a distributed setting a consequence of the existence of parallel execution opportunities is that resource consumption does *not* give a good indication of the plan response time. This is illustrated in the following example.

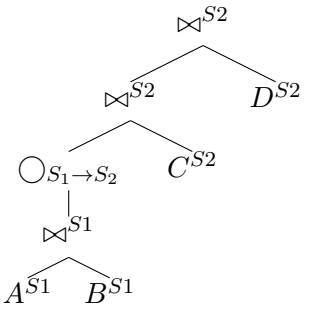


Figure 15: Sequential plan

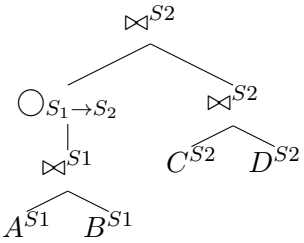


Figure 16: Parallel plan

Figure 17: Parallel-Sequential Plan Example

The following notation is used throughout this example.

- s : the plan in Figure 15.
- p : the plan in Figure 16.
- y_i : the number of disk I/Os performed by plan i where $i \in \{p, s\}$
- x : the number of pages transfered over the network.
- t_d : the time required to read/write a page to disk.
- t_n : the time required to transfer a page from one site to another across the network.
- a and b : factors used to produce the resource consumption metric.
- $C_{RC}(l)$: The cost of plan l in terms of the resource consumption metric.
- $C_T(l)$: the cost of plan l in terms of response time.
- h : the subplan $A^{S1} \bowtie^{S1} B^{S1}$.

Consider establishing the cost of the plans given in Figure 17 using a cost metric based on resource consumption. For this we require two factors a and b to convert the number of I/Os and network page transfers respectively into a common comparable result. These factors could possibly be based on an expense that is related to using the particular resource. The formulas for the resource consumption cost of each plan can be seen below (1 and 2). Note that both plans involve

transferring the same amount of data between sites hence we have the common xb term present. Suppose that the subplan $A^{S1} \bowtie^{S1} B^{S1}$ performs less I/Os than the subplan $C \bowtie D$. We consider the situation where inequality 4 holds i.e. where $C_{RC}(s) < C_{RC}(p)$. A cost model based on resource consumption would determine that plan s was a better execution plan than p as s uses less resources.

$$C_{RC}(s) = ay_s + xb \tag{1}$$
$$C_{RC}(p) = ay_p + xb \tag{2}$$
$$C_{RC}(h) = ay_h \tag{3}$$
$$C_{RC}(s) < C_{RC}(p) < C_{RC}(s) + C_{RC}(h) \tag{4}$$

Inequality 4 implies the following.

$$y_s < y_p < y_s + y_h \tag{5}$$

The cost of each plan based on a response time cost model is shown below. To determine the cost of a plan we simply multiply the number of I/Os and network page transmissions by the time taken to read/write a page and transmit a page respectively.

$$\begin{aligned}
C_T(s) &= y_s t_d + x t_n \\
C_T(h) &= y_h t_d \\
C_T(p) &= y_p t_d + x t_n - C_T(h) \\
&= y_p t_d + x t_n - y_h t_d \\
&= (y_p - y_h) t_d + x t_n \\
&< y_s t_d + x t_n \qquad (by\ 5) \\
&= C_T(s)
\end{aligned}$$

In plan p it is possible to perform $A \bowtie B$ and $C \bowtie D$ concurrently as the joins are performed at different sites $S1$ and $S2$. This implies that the cost of plan p will consist of the maximum amount of time required to complete both joins $A^{S1} \bowtie^{S1} B^{S1}$ and $C^{S2} \bowtie^{S2} D^{S2}$ in addition to the network transfer time and the final join time. The response time of p can therefore be obtained by taking the minimum time ($C_T(h)$) away from the sequential execution time of p. In this

particular example we find that even though plan s has a lower resource consumption cost ($C_{RC}(s) < C_{RC}(p)$) plan p has a lower cost with respect to response time ($C_T(p) < C_T(s)$). In this thesis we focus on the problem of identifying the plan with the lowest response time hence any cost model has to be able to identify possible parallelism opportunities within a given plan and produce the minimum response time accordingly. As we shall see in section 4 this problem is an instance of the NP-hard precedence constrained scheduling problem.

4 Cost Model For Distributed Evaluation

In the previous section we discussed some of the main conceptual problems encountered when building an optimizer for a distributed DBMS. One issue highlighted was that determining the response time of a query plan in a distributed database system is considerably more complex than in a centralized system because of the opportunity for parallelism. A cost model for a distributed system must accurately model the sources and deterrents of parallelism in order to produce an accurate estimate of the response time for a given plan. In this thesis we focus on modelling data independent inter-operator parallelism and as a result we make the assumption that intermediate results are always materialized. We consider both parallelism deterrents resource contention and operator dependencies in our model. We consider only plans that consist of join, receive, send and relation scan operators. In order to reduce the complexity of the model we assume that each site in the system consists of a single processor that is only capable of performing a single operation at a time. The solution proposed here involves the *plan transformation* phase and the *scheduling* phase.

4.1 Plan Transformation Phase

In the plan transformation phase we transform the given plan into a form that allows data dependencies and some degree of resource contention to be modelled. We first identify connected subsections of a given plan that contain operators annotated with the same execution site. By grouping operators with a common execution site into *tasks* we are able to more easily identify resource contention and dependencies. The goal of the plan transformation phase is to break a plan into a set of tasks and to model dependencies between tasks using a *task tree*. The task tree can then be used in the subsequent scheduling phase to ascertain the response time. The task tree used in this paper is a variation of the task tree described in [7], which was used to model parallelism between operations in a *parallel* database system.

Let us illustrate the plan transformation phase using an example. Suppose we are presented with the problem of determining the response time of the plan given in Figure 18. Notice that each node is annotated with a site identifier, i.e. the IP address. We first need to identify tasks that consist of operators with a common execution site. By considering a number of operators as a single task we reduce the complexity of the subsequent scheduling phase. Tasks can be identified by iterating over the given plan using a post order traversal and placing *delimiters* ($\boxtimes$) between operators using the rules specified below. Delimiters are used to signal where a

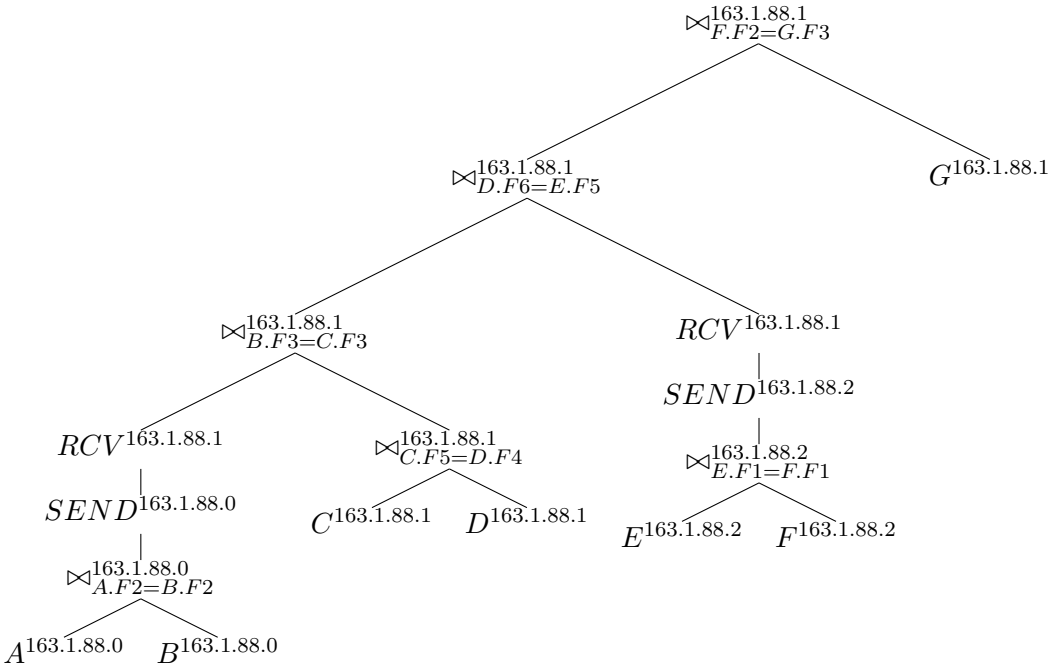


Figure 18: Example plan

task begins and ends.

- If the current node is a SEND or RCV operator place a delimiter inbetween this node and the child node.
- If the current node is a $\bowtie$ operator and the site of at least one child node does not match the site of this node then place a delimiter between this node and *each* child node.
- If the current node is a $\bowtie$ operator and one of the child nodes is a RCV node then place a delimiter between this node and *each* child.

The operator tree obtained as a result of applying these rules to the plan in Figure 18 can be seen in Figure 19. The motivation for the delimiter placement rules regarding the SEND and RCV operator is as follows. Most network transmission protocols use acknowledgement based Automatic Repeat Request (ARQ) protocols [6]. During the transmission process these protocols enforce that both the sender and receiver remain active until the transmission is complete. By dedicating a task to RCV and dedicating a separate task to SEND we can stipulate that these tasks must start and end at the same time. This stipulation is an example of a *task*

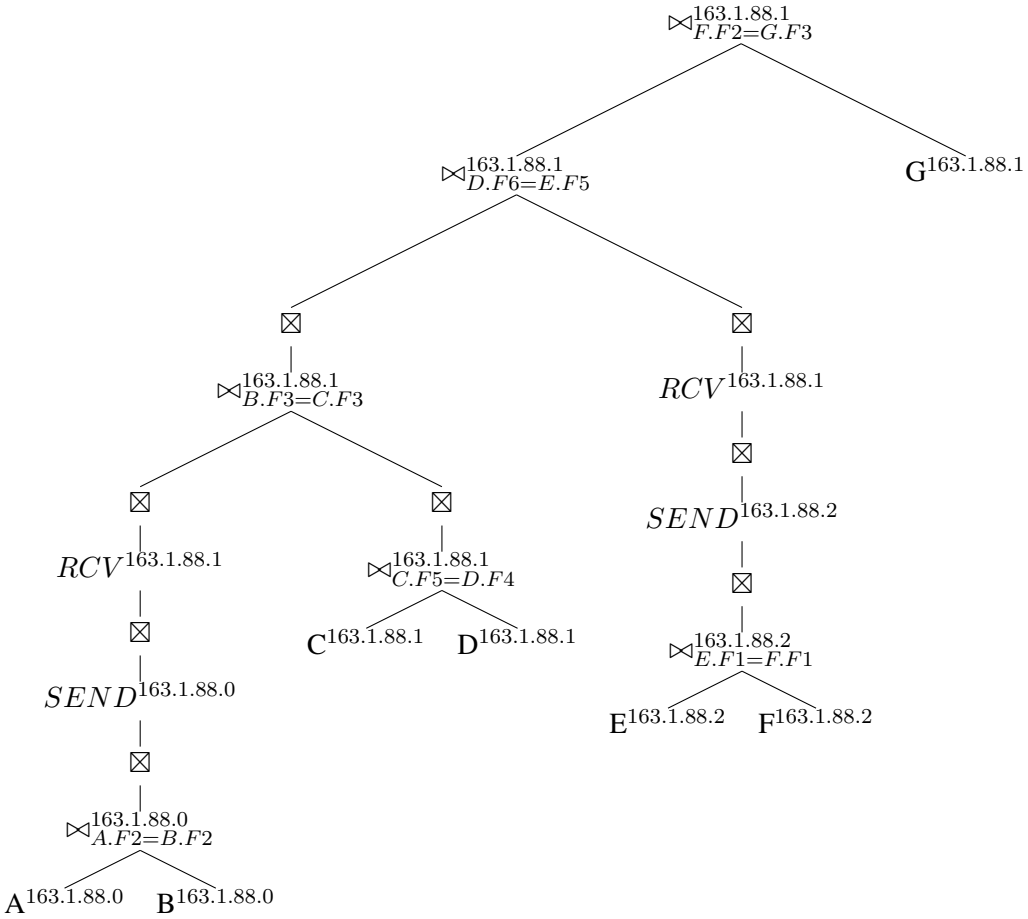


Figure 19: Plan with delimiters

dependency. It would not be possible to have a single task involving a SEND and RCV operation as they occur at different sites. In the event where we have a join operation that has two child operators annotated with the same site as itself we can consider the join along with its children as a single task.

A join cannot be performed until its input is available. Consider the case where we have a task tk containing a join operator o and we have tasks tk_{outer} and tk_{inner} containing the outer and inner child of o respectively. We say that there exists a task dependency between tk and tk_{outer} and tk and tk_{inner}. This dependency stipulates that tasks tk_{outer} and tk_{inner} must be executed before the execution of tk can begin. However, no such task dependency exists between tk_{outer} and tk_{inner}. We also

have a similar situation involving a SEND task tk_{SEND} and the task containing its child tk_{child}, i.e. we cannot send data before the data is made available to the SEND operator. In this case there exists a task dependency between tk_{SEND} and tk_{child}. We can present tasks along with their dependencies in a task tree, where nodes represent tasks and edges represent dependencies between tasks. A task cannot be executed until all children tasks have been completed. The only exception to this case involves tasks containing SEND (tk_{SEND}) and RCV operators (tk_{RCV}). tk_{SEND} is a child of tk_{RCV} but these tasks must be executed concurrently. This is indicated in the task tree by the presence of the $\|$ symbol between tasks. The task tree obtained from the plan in Figure 19 is shown in Figure 20. Each task node in the task tree is labelled with an identifier integer, an execution site and the response time required to complete the task. By ensuring the tasks contain only operators of common execution sites we reduce the complexity of the problem of finding the response time of individual tasks to that of the centralized database system case.

The response time of a particular task can be given by the sum of the response times for each operator residing in the task. In order to determine the response time cost (C_T) of an individual operator we consider the number of I/Os $n_{I/O}$ performed and the number of bytes n_b sent across the network. Assume that we have system specific time constants t_d and t_n that specify the time taken to read/write a disk page and the time taken to send a byte across the network respectively. Using these constants we can estimate the total response time of each operator o by the following equation.

$$C_T(o) = n_{I/O} \cdot t_d + n_b \cdot t_n \tag{6}$$

Each task from Figure 20 is shown along with their respective cost in Figure 1.

4.1.1 Scheduling Phase

Once the plan transformation phase is complete we are faced with the problem of scheduling a task tree containing m tasks $\{tk_1, ..., tk_m\}$ on s sites. Let the site function $\zeta(tk_i)$ give the site at which a particular task tk_i is to be executed. Then each task tk_i is required to be executed *only* at $\zeta(tk_i)$ and each site is considered only able to execute a single task at a time. Tasks are considered to be pre-emptive, i.e. every task tk_i must be executed for a period of $t(tk_i)$ to completion once

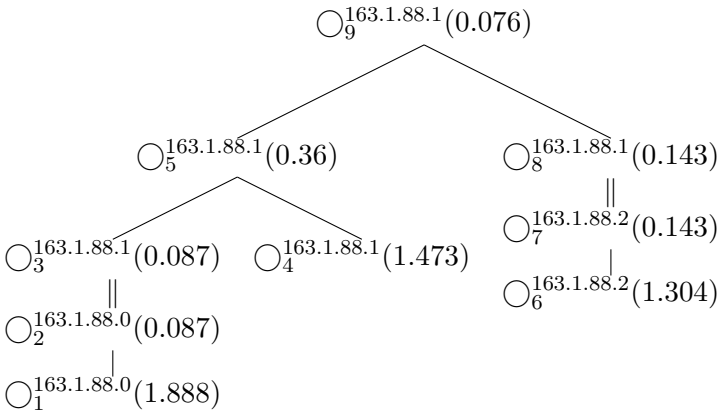


Figure 20: Task tree obtained from the plan in Figure 19.

execution of the task begins. Here $t(tk_i)$ denotes the duration of the execution of the task tk_i. The task tree specifies *precedence constraint* information between tasks that indicates the order in which tasks must be executed. We wish to obtain a schedule $S = \{e_1, e_2, ..., e_m\}$ of m entries of the form $e_i = (tk_i, t_s, t_f)$. There exists an entry in the schedule for each task detailing the start t_s and finish t_f time. Clearly $t_f - t_s$ should equal $t(tk_i)$ for all schedule entries. Once the schedule is obtained the response time is equal to the maximum t_f value of all elements in S. This problem is an instance of the NP-hard multiprocessor precedence constrained scheduling problem [3] and hence no algorithm has been discovered to obtain the optimal schedule in polynomial time.

The process of determining the cost of an execution plan is performed frequently by the enumeration algorithm during optimization. By adopting a scheduling algorithm that is optimal but that has a high time complexity the enumeration algorithm would become unusable. Instead of investing considerable effort into obtaining perfect schedules we opt for a more practical solution that produces good schedules with a worst case running time of $\mathcal{O}(m^2)$.

We create s bins $b_1, b_2, ..., b_s$ where bin b_i stores schedule entries for tasks intended to execute at site s_i. Schedule entries can be placed anywhere in the appropriate bin where a sufficient space in time exists and where precedence constraints are not violated. A post order traversal of the task tree is conducted to ensure that tasks are visited in an order conforming with the existing precedence constraints. When a task tk_i is encountered a schedule entry e_i is created for the task, which is placed in the bin corresponding to site $\zeta(tk_i)$. The position at which e_i is inserted into the bin is determined primarily on the maximum t_f value (t_{fmax}) of the schedule entries

Task Number	Task	Duration
9	$\bowtie^{163.1.88.1}_{F.F2=G.F3}$ $\bowtie^{163.1.88.1}_{D.F6=E.F5}$ $G^{163.1.88.1}$	0.076
5	$\bowtie^{163.1.88.1}_{B.F3=C.F3}$	0.36
3	$RCV^{163.1.88.1}$	0.087
2	$SEND^{163.1.88.0}$	0.087
1	$\bowtie^{163.1.88.0}_{A.F2=B.F2}$ $A^{163.1.88.0}$ $B^{163.1.88.0}$	1.888
4	$\bowtie^{163.1.88.1}_{C.F5=D.F4}$ $C^{163.1.88.1}$ $D^{163.1.88.1}$	1.473
8	$RCV^{163.1.88.1}$	0.143
7	$SEND^{163.1.88.2}$	0.143
6	$\bowtie^{163.1.88.2}_{E.F1=F.F1}$ $E^{163.1.88.2}$ $F^{163.1.88.2}$	1.304

Table 1: Tasks

of the children tasks of t_i. If t_i has no children then e_i can be placed at the first available space of b_i. Otherwise e_i must be placed at an available start time t_s such that $t_s \geq t_{fmax}$. In the event where we encounter a pair of tasks corresponding to a SEND-RCV pair each schedule entry is placed in their corresponding bin at the *same* first available position (for both sites) after the maximum finish time of the child of the SEND task schedule entry. For example, tasks 7 and 8 in Figure 21 correspond to a SEND-RCV pair of tasks. Even though task 7 (SEND task) can be executed immediately after task 6 the receiving site is busy executing task 4 and so is not available to receive. Task 7 must therefore be pushed later in the schedule to a time where the receiving site is available.

Response time is considered to be the most important measure of the cost of a plan in this thesis. However, in the situation where we are presented with two plans with equal response time the system *utilization* should be taken into account. The system utilization factor gives an indication of the amount of system resources that are used by a given plan. Let t_{total} be the sum of the durations of individual tasks

in a schedule. If there are s sites in the system and the plan has a response time of t_{resp} the system utilization factor (μ) for plan p is given as follows.

$$\mu(p) = \frac{t_{total}}{s \cdot t_{resp}} \tag{7}$$

The plan with the lower system utilization factor is favourable and is determined to have a lower cost with respect to the response time cost model. The final schedule of the task tree in Figure 1 can be seen in Figure 21. From this schedule we can deduce that the response time of the original plan in Figure 18 is about 2.41 seconds and the utilization factor is 0.767. In terms of the schedule diagram the utilization factor is the ratio of the total area occupied by tasks to the total available area.

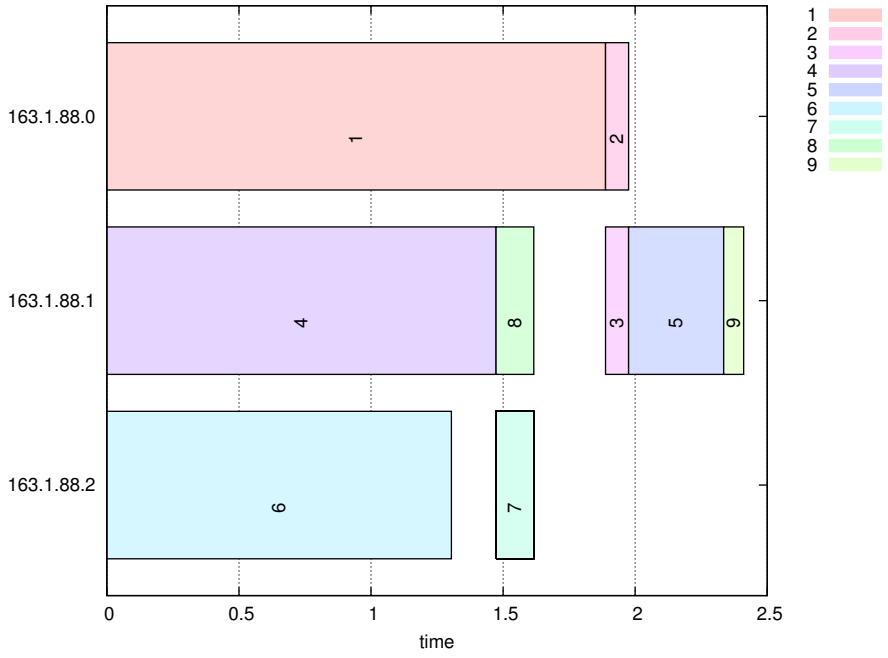


Figure 21: Schedule produced from task tree in Figure 20

5 Centralized Optimization Algorithms

In the previous section we presented a cost model capable of producing the response time of a given plan. This cost model will be used in conjunction with the enumeration algorithms presented in this and the following section. In this section we present the DPccp (Dynamic programming connected subset complement pair) optimization algorithm [21] that applies dynamic programming to the join graph of a query in order to enumerate over the minimal number of joins possible without considering cross products. DP (Figure 12) and DPccp have the same worst case running time of $\mathcal{O}(s^3 \cdot 3^n)$ but in practise DPccp can cause considerable savings in optimization time when optimizing chain and cycle queries. DPccp is used as part of the multilevel optimization algorithms proposed in the next section.

We also present the Iterative Dynamic Programming algorithm IDP1 [9] that combines heuristics with dynamic programming in order to overcome the space complexity problem of DP. We combined this algorithm with DPccp to produce the IDP1ccp algorithm. This algorithm was considered in [9] but at the time of publication in 2000 DPccp had not yet been produced. Consequently, only the optimization of small simple queries was considered in [9], which presumably allowed the authors to create a version of IDP1ccp using a degree of code hardwiring. We feel that it is important to present the general IDP1ccp algorithm as in practise it can produce considerable optimization time savings when compared with IDP1. IDP1ccp is used as a benchmark to the multilevel optimization algorithms proposed in this paper.

Throughout this section and the next we use the following chain query involving the relations R_1, R_2, R_3 and R_4 to demonstrate each of the optimization algorithms presented.

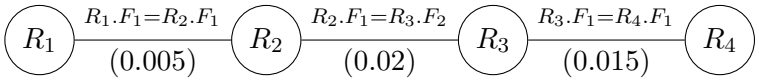


Figure 22: Example Join Graph

5.1 IDP1 : Iterative Dynamic Programming Algorithm for Query Optimization

The main drawback of the DP algorithm (section 2.3.2) is that a large enough query will cause the algorithm to completely fill the available main memory. In this event

the optimizing program may crash or the operating system may suffer from high disk paging causing performance to degrade significantly. In practise the size of the query that leads to this situation is fairly small, which means that DP is not a suitable enumeration algorithm to use when the optimization of fairly large (involving > 20 relations) queries is required. The Iterative Dynamic Programming (IDP) class of enumeration algorithms was proposed to primarily address this space complexity issue of DP. IDP combines dynamic programming with a greedy heuristic in order to overcome the memory restriction. Multiple variants of IDP exists but we choose to present the standard best plan variant of IDP1 below as this gives the fundamental principles of IDP. We then give the balanced best row IDP1 algorithm as this proved to give the best quality plans during the experiments in [9].

Consider the situation where we wish to optimize a query involving the relations $R = \{R_1, R_2, ..., R_n\}$ in a centralized database system. Let us assume that the memory available is sufficient only to store all plans up to k-way plans after pruning, i.e. access plans, 2-way plans, 3-way plans, ..., k-way plans, where $k < n$. If DP was the enumeration algorithm used we would progress to the stage where k-way plans are built and then exhaust the available memory by attempting to construct $k+1$-way plans. Instead of creating $k+1$-way plans IDP1 breaks out of the dynamic programming phase and applies a greedy heurstic in order to choose the k-way plan P with the lowest value with respect to an evaluation function $eval$ [9]. All plans in the optPlan structure that contain any relation involved in P are then removed from memory. From this point onwards in the algorithm P is considered to be an access method for a temporary relation $\mathcal{T}$. Let the set of relations involved in $\mathcal{T}$ be denoted by $R_{\mathcal{T}}$. The dynamic programming phase is then entered again with the available relations $(\{\mathcal{T}\} \cup R) - R_{\mathcal{T}}$. The process of performing DP and then breaking out to perform a greedy heurstic continues until the relation set R has a size no more than k. In this case there is enough memory to carry out the DP phase to completion where the final plan is produced. In a centralized system IDP1 has a time complexity of $\mathcal{O}(n^k)$. In a distributed setting consisting of s sites IDP1 has a time complexity of $\mathcal{O}(s^3 \cdot n^k)$ and space complexity of $\mathcal{O}(s \cdot n^k + s^3)$ [9].

5.1.1 Example : Standard-best plan IDP1

Consider the example where we wish to use IDP1 to optimize the query in Figure 22. Assume that the available memory can hold all plans up to 3-way plans and no more, i.e. $k = 3$. An example of the retained plans in the optPlan structure after pruning for the first DP phase can be seen in Figure 23. Once the 3-way plans are

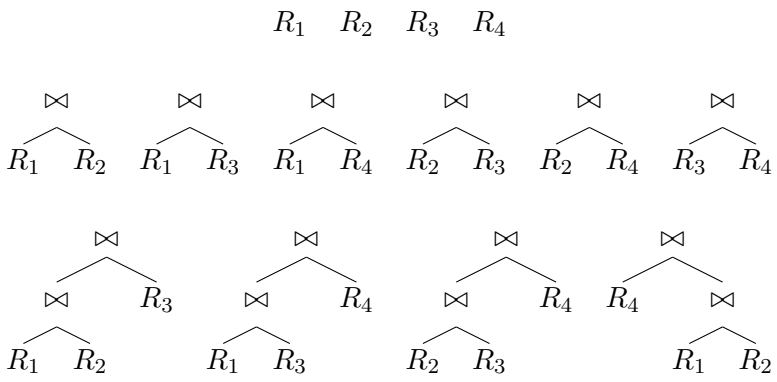


Figure 23: Stored plans in first DP phase

produced we greedily choose the 3-way plan P with the lowest $eval$ cost. Let P correspond to the plan in Figure 24 and let the relations R_1, R_2 and R_3 be replaced by the temporary relation $\mathcal{T}$. All plans containing a relation involved in P are then removed from memory. In this instance this includes all plans except for P and the access method R_4. The newly considered access plans are shown in Figure 25 along with an example of the final result obtained from the second DP phase. By substituting the plan P for $\mathcal{T}$ we obtain the final plan in terms of the original relations (shown in Figure 26).

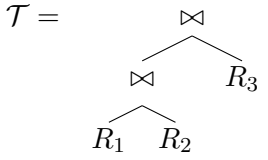


Figure 24: Chosen plan using greedy heurstic with $eval$ function

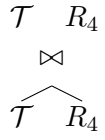


Figure 25: Second DP phase

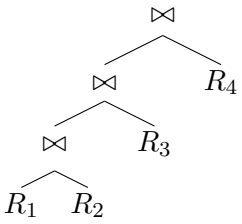


Figure 26: Plan obtained using standard-best plan IDP1

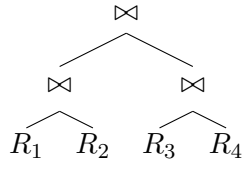


Figure 27: Optimal plan

Consider the situation where the optimal plan is the bushy plan shown in Figure 27. Due to the value of k this optimal plan can never be encountered using the standard IDP1 algorithm in this situation. In the *balanced* IDP1 variant the building block plans have a size restriction placed upon them that makes it possible for bushy plans to be produced regardless of the k value. The criteria governing the size of building block plans is as follows [9].

- The selection subplans must contain an even number of relations.
- The selection subplans must contain a number of relations $\leq \lceil \frac{d}{2} \rceil$, where d is the number of access relations available at the start of the current DP phase.

In the final DP phase where the number of relations remaining is $\leq k$ we simply use the value of k for the size of building blocks. In the *best row* variant when we determine the plan P with the lowest $eval$ value instead of just choosing plan P as the optPlan entry for $\mathcal{T}$ we copy all plans involving the same relations as P into the optPlan entry for $\mathcal{T}$. This means that in the case where interesting orders occur we do not simply choose to keep the plan with the lowest $eval$ value, which means that we have a greater chance of obtaining the optimal result. The balanced-best row IDP1 algorithm is shown below.

Input: The set of relations $R = \{R_1, R_2, ..., R_n\}$ involved in the query and the maximum building block size $k > 1$.

Output: The best query execution plan found.

Algorithm IDP1$(R = \{R_1, R_2, ..., R_n\}, k)$

```
1  for i ← 1 to n do {
2     opt-plan({R_i}) ← ACCESS-PLANS({R_i})
3     PRUNE-PLANS(opt-plan({R_i}))
4  }
5  toDo ← R
6  while |toDo| > 1 do {
7     b ← BALANCED-PARAMETER(|toDo|, k)
8     for i ← 2 to b do {
9           for all S ⊂ R such that |S| = i do {
10             opt-plan(S) ← ∅
11             for all O ⊂ S where O ≠ ∅ do {
12                   opt-plan(S) ← opt-plan(S)∪
                           JOIN-PLANS(opt-plan(O), opt-plan(S − O))
13                   PRUNE-PLANS(opt-plan(S))
14             }
15          }
16    }
17    find P, N with P ∈ opt-plan(N), N ⊂ toDo, |N| = b such that
          eval(P) = min{ eval(P') | P' ∈ optPlan(W), W ⊂ toDo, |W| = b}
18    generate new symbol: T
19    opt-plan({T}) ← opt-plan(N)
20    toDo ← toDo − N ∪ {T}
21    for all O ⊂ N do {
22          delete(opt-plan(O))
23    }
24 }
25 FINALIZE-PLANS(opt-plan(R))
26 PRUNE-PLANS(opt-plan(R))
27 return opt-plan(R)
```

Input: The number of relations d and maximum block size k.
Output: The balanced block parameter.

Algorithm BALANCED-PARAMETER(d, k)

```
1  b ← d
2  if b > k do {
3      b ← ceil(b/2)
4      if b ≡ 1 mod 2 do {
5          b ← b − 1
6      }
7      b ← min(b, k)
8  }
9  return b
```

5.1.2 Example : Balanced-best plan IDP1

Again, consider the example where the available memory is only able to store all k-way plans up to $k = 3$. We choose the balanced parameter $b = 2$ as it is the maximum even value that is $\leq \lceil \frac{n}{2} \rceil$ where $n = 4$. Instead of building all plans up to 3-way plans in the first DP phase, as standard-best plan IDP1 does, we only build up to 2-way plans. An example of the contents of the optPlan structure can be seen in Figure 28.

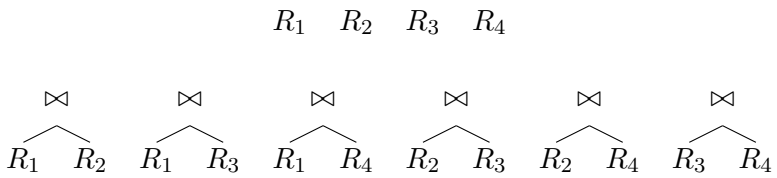


Figure 28: Stored plans in first DP phase

We then choose the plan with the lowest *eval* value, which from the previous case was the plan P containing R_1 and R_2. All plans involving either of these relations are removed from memory and the temporary relation $\mathcal{T}$ is used to represent P. The relations waiting to be optimized are now $\mathcal{T}$, R_3 and R_4, which implies that the balanced parameter for the second DP phase is 2. The possible contents of the optPlan structure after the second DP phase (after pruning) is shown in Figure 29.

We then choose the 2-way plan with the lowest $eval$ value. Let P' be this plan involving R_3 and R_4. We remove all plans from memory containing either of these

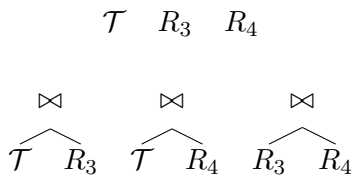


Figure 29: Stored plans in second DP phase

relations and then we add the temporary relation $\mathcal{S}$ in place of P'. The final DP phase will then only contain the temporary relations $\mathcal{T}$ and $\mathcal{S}$. The possible final plan (in terms of temporary relations) can be seen in Figure 30. The final plan involving the base relations can be seen in Figure 31.

Figure 30: Plan obtained using balanced-best plan IDP1

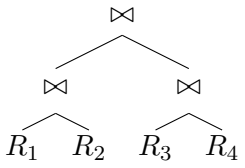


Figure 31: Final plan involving base relations

5.2 DPccp : Dynamic Programming connected subset complement-pair Algorithm for Query Optimization

Both DP and IDP consider a query as a set of relations $R = \{R_1, R_2, ..., R_n\}$ and as a result they consider all possible combinations of joins (in keeping with the principle of optimality). This type of enumeration is suitable when presented with a query possessing a clique join graph. However, in the event where we are presented with a chain or cycle query this enumeration can cause many cross products to be considered. Cross products are an expensive operation and are rarely found in the optimal plan hence we would like to enumerate over only combinations of relations without considering cross products. This kind of enumeration is provided by the DPccp algorithm [21], which we discuss below.

5.2.1 Definitions

Consider a query involving the relations $R = \{R_1, R_2, ..., R_n\}$ that has a join graph $G = (V, E)$ and let S be a subset of R. Then S gives rise to a subgraph $S_G = (V', E')$ of G containing vertices $V' = \{v_1, ..., v_{|S|}\}$, where each vertex corresponds to a relation in S. The edge set E' of S_G is given by the set $\{(v_1, v_2) \mid v_1, v_2 \in V', (v_1, v_2) \in E\}$. We say that S is a connected subset if it induces a connected subgraph S_G. The number of connected subsets/subgraphs of a query involving n relations is denoted as $\#csg(n)$.

Let S_1 and S_2 be nonempty connected subsets of R that induce the connected subgraphs S_{1G} and S_{2G}. We say that (S_1, S_2) is a csg-cmp-pair if the following holds.

- $S_1 \cap S_2 = \emptyset$
- $\exists v_1 \in S_{1G}$ and $\exists v_2 \in S_{2G}$ such that $(v_1, v_2) \in E$.

The term *cmp* in the name *csg-cmp-pair* is an abbreviation of *complement*, which emphasises that the two sets S_1 and S_2 are disjoint [21]. Let *cpp* denote the set of csg-cmp-pairs. Note that if (S_1, S_2) $\in$ *ccp* then (S_2, S_1) $\in$ *cpp*. Let the size of the set *ccp* be denoted by $\#ccp$. Note that $\#cpp$ includes symmetric pairs. This value provides a lower bound on the number of joins that should be considered during the enumeration process. In the following section we give formulas for the number of connected subgraphs ($\#csg(n)$) and the number of csg-cmp-pairs ($\#ccp(n)$) for the join graph structures chain, cycle, star and clique.

5.2.2 $\#csg$ and $\#ccp$ formulas

In this section we present the formulas for $\#csg(n)$ and $\#ccp(n)$ as given in [21]. They can be seen in Figure 32. Notice that for star and clique queries $\#ccp(n)$ is exponential with n, while chain and cycle queries have a $\#ccp(n)$ given by a polynomial with degree 3. Let us consider the example where we wish to enumerate over csg-cmp-pairs of a chain and clique query involving 10 relations. $\#ccp$ for a chain query is 330, while $\#ccp$ for a clique query is 57002. The DP algorithm (Figure 12) enumerates over the same number of plans (equal to $\#ccp$ for a clique query) regardless of the query structure. This implies that by using the DP algorithm with a chain query we enumerate over 56672 cross products that will not be present in the optimal plan. The DPccp algorithm ensures that the enumeration lower bound of $\#ccp(n)$ is met for all join graphs and hence we do not consider any unnecessary cross products.

Join Graph Structure	$\#csg(n)$	$\#ccp(n)$
Chain	$\frac{n(n+1)}{2}$	$\frac{n^3-n}{3}$ †
Cycle	$n^2 - n + 1$	$n^3 - 2n^2 + n$
Star	$2^{n-1} + n - 1$	$(n-1)2^{n-2}$
Clique	$2^n - 1$	$3^n - 2^{n+1} + 1$

Figure 32: $\#csg$ and $\#ccp$ formulas

5.2.3 Algorithm Description

The DPccp algorithm enumerates all *unique* csg-cmp-pairs in such a manner as to be compatible with dynamic programming. This means that when we consider the csg-cmp-pair (S_1, S_2) then the csp-cmp-pairs consisting of subsets of S_1 and S_2 will have already been considered. With this enumeration criteria met the DPccp algorithm simply enumerates over all csg-cmp-pairs (S_1, S_2) and considers the possible joins between the best plans of S_1 and S_2. The algorithm is shown below. Note that the access-plans, prune-plans, join-plans and finalize-plans have the same implementation used by the classic DP algorithm (Figure 12). In the event where we wish this algorithm to be applicable in the distributed setting we perform the simple alterations to these methods as suggested in [9].

† The #ccp formula in [21] for chain graph is incorrect. The proof of the formula given here can be seen in Appendix C.

Input: Connected join graph G consisting of vertices $V = \{v_1, v_2, ..., v_n\}$ and edges E.

Output: The best query execution plan found (without considering cross products).

Algorithm DPCCP(G)
1 **for** $i \leftarrow 1$ **to** n **do** $\{$
2 $\quad opt\text{-}plan(\{v_i\}) \leftarrow$ ACCESS-PLANS$(\{v_i\})$
3 $\quad$ PRUNE-PLANS$(opt\text{-}plan(\{v_i\}))$
4 $\}$
5 ENUMERATE-CSG(G, A)
6 **for all** $S_1 \in A$ **do** $\{$
7 $\quad$ ENUMERATE-CMP(G, S_1, B)
8 $\quad$ **for all** $S_2 \in B$ **do** $\{$
9 $\quad\quad S \leftarrow S_1 \cup S_2$
10 $\quad\quad opt\text{-}plan(S) \leftarrow opt\text{-}plan(S) \cup$ JOIN-PLANS$(opt\text{-}plan(S_1), opt\text{-}plan(S_2))$
11 $\quad\quad opt\text{-}plan(S) \leftarrow opt\text{-}plan(S) \cup$ JOIN-PLANS$(opt\text{-}plan(S_2), opt\text{-}plan(S_1))$
12 $\quad\quad$ PRUNE-PLANS$(opt\text{-}plan(\{S\}))$
13 $\quad\}$
14 $\quad B \leftarrow \emptyset$
15 $\}$
16 FINALIZE-PLANS$(opt\text{-}plan(V))$
17 PRUNE-PLANS$(opt\text{-}plan(V))$
18 **return** $opt\text{-}plan(V)$

The DPccp algorithm adopts the same first phase (lines 1-4) and final phase (lines 16-18) as DP. The main difference occurs in the second phase (lines 5-15). Note that we enumerate through the *unique* csg-cmp-pairs so it is necessary to explore the joining of plans of subgraphs in both available orders (lines 10 and 11). We now proceed by explaining how to enumerate through the set of connected subgraphs, i.e. the enumerate-csg procedure description, and then we discuss how to generate all complement subgraphs for a given connected subgraph, i.e. the enumerate-cmp procedure description.

5.2.4 Enumerating Connected Subsets

Let us make the following notation. Let $G = (V, E)$ be a given join graph. For $v \in V$ the *neighbourhood* $\mathcal{N}$ of v is defined as follows.

$$\mathcal{N}(v) = \{v' \mid (v, v') \in E\}$$

For a given connected subset S that induces the connected subgraph $S_G = (V', E')$ the neighbourhood of S is defined as follows.

$$\mathcal{N}(S) = (\cup_{v \in V'} \mathcal{N}(v)) - V'$$

Consider a connected subset S and the sets $N \subseteq \mathcal{N}(S)$. Then $S \cup N$ is a connected subset. This indicates that a possible method of generating all connected subsets is as follows. First we identify each individual vertex $\{v_i\}$ as a connected subgraph. We can then enumerate through all $N' \subseteq \mathcal{N}(\{v_i\})$ and recursively explore the connected subset $N' \cup \{v_i\}$. The main problem with this method is that we consider duplicates. In order to overcome this problem each vertex in the join graph is numbered using breadth first numbering starting from any vertex. Let the vertices in the join graph G be the set $V = \{v_1, v_2, ..., v_n\}$, where vertex v_i is the ith vertex encountered using a breadth first vertex enumerator starting from v_1. Then to avoid duplicates only subsets S obtained by recursively exploring the neighbourhood of $\{v_i\}$ are considered such that all vertices in S have indices $> i$. In order to enforce this restriction we require the following notation $\mathcal{B}_i = \{v_j \mid j \leq i\}$ [21]. The enumerate-csg algorithm is given below.

Input: Connected join graph G consisting of vertices $V = \{v_1, v_2, ..., v_n\}$ and edges E. A stores the output therefore A holds the set of all connected subgraphs of G after the algorithm terminates.
Precondition: Vertices in G are numbered in a breadth first manner.
Output: Obtains all subsets of V that form a connected subgraph of G.

Algorithm ENUMERATE-CSG(G, A)
1 **for** $i \leftarrow n$ **down to** 1 **do** {
2 $A \leftarrow A \cup \{\{v_i\}\}$
3 ENUMERATE-CSG-REC(G, $\{v_i\}$, $\mathcal{B}_i$, A)
4 }

Input: Connected join graph G consisting of vertices $V = \{v_1, v_2, ..., v_n\}$ and edges E. S is a set of vertices that contain the current subgraph state. X is the prohibited set of vertices that cannot be explored further. This set enables duplicates to be avoided.
A is used to store connected subgraphs of G.
Precondition: Vertices in G are numbered in a breadth first manner.
Output: Recursively obtains all subsets of V that form a connected subgraph of G.

Algorithm ENUMERATE-CSG-REC(G, S, X, A)

```
1  N ← 𝒩(S) − X
2  for all S′ ⊆ N, S′ ≠ ∅, enumerate subsets first {
3     A ← A ∪ {S ∪ S′}
4  }
5  for all S′ ⊆ N, S′ ≠ ∅, enumerate subsets first {
6     ENUMERATE-CSG-REC(G, S ∪ S′, X ∪ N, A)
7  }
```

5.2.5 Example : enumerate-csg

In Figure 33 we can see a partial trace of the enumerate-csg algorithm. We show the recursive calls of the Enumerate-Csg-Rec procedure and values of the variables N and A. We only show the final two loops of Enumerate-Csg (when $n = 2$ and $n = 1$) as this gives a good indication as to how duplicates are avoided. When the Enumerate-Csg-Rec procedure is provided with the $\{R_1\}$ vertex set we see that we have no restriction on the exploration of neighbours. This is due to the prohibited vertex set $\mathcal{B}_1$ being trivally equal to $\{R_1\}$. We can therefore produce all connected subsets containing the vertex R_1. However, when the Enumerate-Csg-Rec procedure is provided with $\{R_2\}$ we see that the prohibited vertex set $\mathcal{B}_i$ is non trivial and is equal to $\{R_1, R_2\}$. This prevents any exploration involving the vertex R_1. As a result all connected subsets involving R_2 are found except the subset $\{R_1, R_2\}$, which is produced by the recursive exploration from vertex R_1 instead.

5.2.6 Enumerating Complement Subsets

In order to generate all csg-cmp-pairs we first need to generate all connected subsets S_1. Then for each S_1 we need to produce all complement subsets S_2. The concept of producing complements given a particular subset S_1 is the same as the

$A \leftarrow A \cup \{R_1\}$
Enumerate-Csg-Rec(G, $\{R_1\}$, $\{R_1\}$, A)
    $N \leftarrow \{R_2\}$
    $A \leftarrow A \cup \{R_1, R_2\}$
    Enumerate-Csg-Rec(G, $\{R_1, R_2\}$, $\{R_1, R_2\}$, A)
        $N \leftarrow \{R_3\}$
        $A \leftarrow A \cup \{R_1, R_2, R_3\}$
        Enumerate-Csg-Rec(G, $\{R_1, R_2, R_3\}$, $\{R_1, R_2, R_3\}$, A)
            $N \leftarrow \{R_4\}$
            $A \leftarrow A \cup \{R_1, R_2, R_3, R_4\}$
            Enumerate-Csg-Rec(G, $\{R_1, R_2, R_3, R_4\}$, $\{R_1, R_2, R_3, R_4\}$, A)
                $N \leftarrow \emptyset$

$A = \{\{R_1\}, \{R_1, R_2\}, \{R_1, R_2, R_3\}, \{R_1, R_2, R_3, R_4\}\}$

$A \leftarrow A \cup \{R_2\}$
Enumerate-Csg-Rec(G, $\{R_2\}$, $\{R_1, R_2\}$, A)
    $N \leftarrow \{R_3\}$
    $A \leftarrow A \cup \{R_2, R_3\}$
    Enumerate-Csg-Rec(G, $\{R_2, R_3\}$, $\{R_1, R_2, R_3\}$, A)
        $N \leftarrow \{R_4\}$
        $A \leftarrow A \cup \{R_2, R_3, R_4\}$
        Enumerate-Csg-Rec(G, $\{R_2, R_3, R_4\}$, $\{R_1, R_2, R_3, R_4\}$, A)
            $N \leftarrow \emptyset$

$A = \{\{R_1\}, \{R_1, R_2\}, \{R_1, R_2, R_3\}, \{R_1, R_2, R_3, R_4\}, \{R_2\}, \{R_2, R_3\}, \{R_2, R_3, R_4\}\}$

Figure 33: Partial Enumerate-Csg trace

concept of producing connected subsets on a restricted graph. This indicates that we are able to reuse the enumerate-csg-rec procedure with the correct parameters in order to obtain all complements of a given connected subset. As with enumerating all connected subsets we use the breadth first numbering to avoid generating duplicate pairs. This is done by restricting the complements S_2 to contain only vertices v_j such that $j > i$ where i is the index of *any* vertex in S_1 [21]. Given a connected subset S_1 denote $min(S_1) = min(\{i \mid v_i \in S_1\})$. The enumerate-cmp algorithm is given below.

Input:	Connected join graph G consisting of vertices $V = \{v_1, v_2, ..., v_n\}$ and edges E. S is a connected subset of G. A is used to store results.
Precondition:	Vertices in G are numbered in a breadth first manner.
Output:	Obtains all connected subsets H such that (S, H) is a csg-cmp-pair.

Algorithm ENUMERATE-CMP(G, S, A)

```
1  X ← B_min(S) ∪ S
2  N ← N(S) − X
3  for all v_i ∈ N by descending i {
4     A ← A ∪ {{v_i}}
5     ENUMERATE-CSG-REC(G,{v_i}, X ∪ (B_i ∩ N), A)*
6  }
```

In the algorithm above we are concerned with identifying the neighbours of S that have indices greater than the minimum index of vertices in S. We iterate over the neighbours meeting this criteria and output each vertex as a singleton complement subset. We then explore the neighbourhood around these vertices recursively using the enumerate-csg-rec procedure with the restriction set $X \cup (\mathcal{B}_i \cap N)$. This restriction set prevents commutative duplicates from being obtained [21].

5.2.7 Example : enumerate-cmp

We show the example of using the enumerate-cmp procedure to generate all connected components of the connected subgraph $\{R_2, R_3\}$. The trace of the algorithm can be seen below. Note that only a single connected component subset is produced ($\{R_4\}$). There is of course another connected component subset of $\{R_2, R_3\}$, namely $\{R_1\}$. This csg-cmp-pair will be part of the csg-cmp-pairs produced in DPccp when the enumerate-cmp procedure is given the input $\{R_1\}$.

*Typo from [21] corrected here.

enumerate-cmp(G, $\{R_2, R_3\}$, A)

$X \leftarrow \{R_1, R_2\} \cup \{R_2, R_3\} = \{R_1, R_2, R_3\}$

$N \leftarrow \{R_1, R_4\} - \{R_1, R_2, R_3\} = \{R_4\}$

$A \leftarrow A \cup \{\{R_4\}\}$

enumerate-csg-rec(G,$\{R_4\}$, $\{R_1, R_2, R_3, R_4\}$, A)

$N \leftarrow \emptyset$

$A = \{\{R_4\}\}$

Figure 34: Short Example : Enumerate-cmp

5.3 IDP1ccp : Iterative Dynamic Programming connected subset-complement pair Algorithm for Query Optimization

In section 5.1 we presented the IDP1 algorithm that combines greedy heuristics with dynamic programming in order to reduce the running time and space complexity of DP, while still producing good plans. In the previous subsection we discussed the DPccp algorithm that is capable of optimizing a query using dynamic programming by enumerating over the minimal number of connected subsets of the given join graph. In this section we present an improved version of the IDP1 algorithm that amalgamates the results from [9] and [21]. The resulting algorithm has a practically superior running time to IDP1 although the worst case running time is still the same (in the situation where a clique query is being optimized). We give this new version of IDP1 the name *IDP1ccp* to indicate that the algorithm is a result of merging IDP1 and DPccp together. IDP1ccp is shown below.

Input: Connected join graph G consisting of vertices $V = \{v_1, v_2, ..., v_n\}$, which correspond to the relations $R = \{R_1, R_2, ...R_n\}$, and edges E. The maximum building block size k.

Output: The best query execution plan found (without considering cross products).

Algorithm IDP1CCP(G, k)

```
1  for i ← 1 to n do {
2     opt-plan({R_i}) ← ACCESS-PLANS({R_i})
3     PRUNE-PLANS(opt-plan({R_i}))
4  }
5  J ← ∅
6  while |V| > 1 do {
7     b ← BALANCED-PARAMETER(|V|, k)
8     ENUMERATE-CSG'(G, A, b − 1)
9     for all S_1 ∈ A do {
10          ENUMERATE-CMP'(G, S_1, B, b)
11          for all S_2 ∈ B do {
12             S ← S_1 ∪ S_2
13             if |S| = b do {
14                   J ← J ∪ {S}
15             }
16             opt-plan(S) ← opt-plan(S) ∪ JOIN-PLANS(opt-plan(S_1), opt-plan(S_2))
17             opt-plan(S) ← opt-plan(S) ∪ JOIN-PLANS(opt-plan(S_2), opt-plan(S_1))
18             PRUNE-PLANS(opt-plan({S}))
19          }
20          B ← ∅
21    }
22    find P, N with P ∈ opt-plan(N), N ∈ J such that
            eval(P) = min{ eval(P') | P' ∈ optPlan(W), W ∈ J}
23    generate new symbol: T
24    opt-plan({T}) ← {P}
25    H ← MERGE-VERTICES(G, N, T)
```

```
26      A ← ∅
27      ENUMERATE-CSG(H, A)
28      for all O ⊂ A do {
29          delete(opt-plan(O))
30      }
31      J ← ∅
32  }
33  FINALIZE-PLANS(opt-plan(V))
34  PRUNE-PLANS(opt-plan(V))
35  return opt-plan(R)
```

Instead of supplying the relations set $R = \{R_1, R_2, ..., R_n\}$ as input (as done with IDP1) we supply the join graph G. The first phase (lines 1-4) and final phase (lines 33-35) are the same as used in most optimization algorithms described so far in this thesis. We no longer have a $toDo$ set as in IDP1 that stored relations waiting to be involved in plans. Instead we use the vertices of the join graph G as an indication of the remaining relations as there exists a one-to-one correspondence between the vertices in V and the relations involved in the query R.

In the second phase of balanced-best plan IDP1 we generate all 2-way, 3-way, ..., b-way plans, break out of the DP loop and choose the best b-way plan P. Here, the parameter b is obtained by applying the procedure balanced-parameter (from section 5.1) to the number of remaining relations and the k value. Let the relations involved in P be denoted by R'. All plans involving relations $r \in R'$ are then deleted from memory and P is then considered to be an access plan for a new temporary relation $\mathcal{T}$. The set of relations being optimized is then updated by removing all relations in R' and adding $\mathcal{T}$. The concept with IDP1ccp is the same except that we use the procedures enumerate-csg and enumerate-cmp to generate the required csg-cmp-pairs instead of enumerating all possible subsets. However, we do not wish to obtain *all* csg-cmp-pairs. We are only concerned with generating pairs (S_1, S_2) such that $|S_1| + |S_2| = b$, i.e. we only want to produce plans up to b-way plans. It is therefore necessary to alter the enumerate-csg, enumerate-csg-rec and enumerate-cmp procedures to seize the recursive exploration once a certain size of subset has been obtained. The changes are minimal and the altered procedures enumerate-csg', enumerate-csg-rec' and enumerate-cmp' can be seen in Appendix A.

The structure J is used to store all enumerated connected subsets containing b vertices. Once we produce all b sized connected subsets we break out of the DP phase and choose the best plan P of the connected subsets in J. We then consider P as an access method of the temporary relation $\mathcal{T}$ as before. The next step (line

25) involves updating the available relations for future DP phases. We need to remove relations involved in P and add $\mathcal{T}$. We no longer have a set of relations R to update but instead have the graph G to update. Updating the relations therefore involves *merging* all vertices corresponding to relations involved in P into a single temporary vertex named $\mathcal{T}$. This merging process is carried out by the *merge-vertices* procedure. Once we have removed the subgraph H we then have to delete all plans involving relations in H. This is achieved by using *enumerate-csg* with H to enumerate all connected subsets of H. For every connected subset $h \in H$ we delete the optPlan entries for h.

Input:	Connected join graph G consisting of vertices $V = \{v_1, v_2, ..., v_n\}$, which correspond to the relations $R = \{R_1, R_2, ...R_n\}$, and edges E. The set of vertices N that will be replaced by the symbol $\mathcal{T}$.
Purpose:	Replaces N in G by a single vertex $\mathcal{T}$ and resets dangling edge references.
Output:	The graph resulting from removing the vertices and edges between these vertices.

Algorithm MERGE-VERTICES$(G, N, \mathcal{T})$

1 $V \leftarrow V - N \cup \mathcal{T}$
2 $E' \leftarrow \{(a, b) \mid a \in N, b \in N, (a, b) \in E\}$
3 **for all** $(a, b) \in \{(c, d) \mid c \in \mathcal{N}(N),\ d \in N, (c, d) \in E\}$ **do**
4 $\quad b \leftarrow \mathcal{T}$
5 }
6 **for all** $(a, b) \in \{(c, d) \mid d \in \mathcal{N}(N),\ c \in N, (c, d) \in E\}$ **do**
7 $\quad a \leftarrow \mathcal{T}$
8 }
9 **return** (N, E')

5.3.1 Example : IDP1ccp

Let us consider applying IDP1ccp with $k = 2$ to the chain join graph $G = (V, E)$ shown in Figure 22. Since $|V| = 4$ the chosen building block size b for the first DP phase will be 2 as this is the largest even value that is less than $\lceil \frac{|V|}{2} \rceil = 2$. We proceed by considering the access plans for the relations R_1, R_2,R_3, R_4 and then considering the 2-way plans that can be created from these access plans. An example of the plans that could be stored in the optPlan structure after the first DP phase is shown in Figure 35.

Assume that the chosen plan after the first DP phase is the plan involving the relations R_2 and R_3 as given in Figure 36. The join graph is then updated by merging

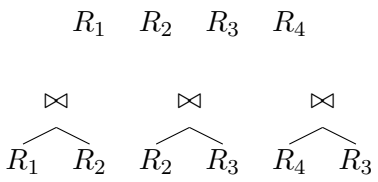


Figure 35: Stored plans in first DP phase (b=2)

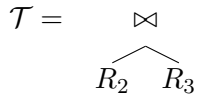


Figure 36: Chosen plan using greedy heurstic with $eval$ function

the vertices corresponding to R_2 and R_3 into a new vertex for the temporary relation $\mathcal{T}$. The resulting join graph can be seen in Figure 37.

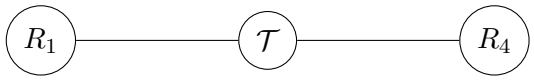


Figure 37: Join Graph after merge phase

We then proceed into the second DP phase with the query relations R_1, $\mathcal{T}$ and R_4. As with the first DP phase, the building block size b is chosen to be 2. As a result, a possible example of the access plans and 2-way plans stored in the optPlans structure is shown in Figure 38. Let the plan with the lowest *eval* value be the plan involving the relations R_4 and $\mathcal{T}$ (shown in Figure 39).

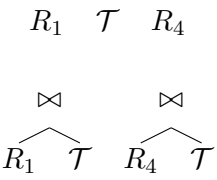


Figure 38: Stored plans in second DP phase (b=2)

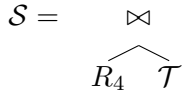


Figure 39: Chosen plan using greedy heurstic with $eval$ function

After performing another merge phase we create the temporary relation vertex $\mathcal{S}$. The resulting join graph is shown in Figure 40. The number of vertices in the join graph is now equal to k therefore in the third DP phase we can produce the final plan. The final plan obtained in terms of temporary relations is shown in Figure 41.

The final plan in terms of base relations can be obtained by substituting optimized plans for the temporary relations to produce the final plan (shown in Figure 42).

Figure 40: Join Graph after 2nd merge phase

Figure 41: Final Plan in terms of temporary relations

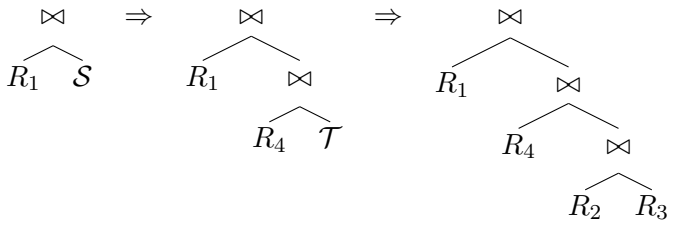


Figure 42: Final plan obtained by IDP1ccp

6 Multilevel Optimization

In the previous section we discussed some existing optimization algorithms, namely IDP1 and DPccp, which we combined to produce the improved algorithm IDP1ccp. When presented with a join graph with a structure similiar to that of a chain or cycle query the time savings of using IDP1ccp over IDP1 can be significant. However, as identified in [9] most of the work exerted by IDP1 is in the first dynamic programming phase. This can be problematic when presented with large queries or queries in a distributed setting involving a moderate number[‡] of execution sites. For example, we find that clique queries involving 100 relations in a centralized system cannot be optimized within 30 seconds using IDP1ccp. Also, in a distributed system consisting of nine execution sites, clique queries involving ≥ 20 relations cannot be optimized within 30 seconds (section 8) when using IDP1ccp. Ideally we would like to have an optimization algorithm that is able to produce good plans even with larger queries or queries in a distributed setting involving a large number of execution sites.

In this section we present a novel class of optimization algorithms to address the shortcomings of IDP1ccp. The primary aim of the algorithms suggested here is to

[‡]By moderate we mean 5-10.

be able to optimize queries that IDP1ccp would not be able to in an optimization time. We begin by presenting the general structure of the class of algorithms, which we name *Multilevel* optimization algorithms. We then proceed by giving a sequential version which is then extended into a distributed version that better utilizes the available resources of a distributed system.

6.1 Multilevel (ML) Framework

When designing an optimization algorithm we would ideally like to use pure dynamic programming and no heuristics as this ensures that we obtain the optimal plan with respect to the incorporated cost model. However, this is not practical for even moderate sized queries or queries involving relations stored at a moderate number of sites. We are therefore faced with the tradeoff of sacrificing plan quality in order to obtain plans for large queries in a reasonable time. The multilevel optimization framework suggested here combines heuristics and dynamic programming but opts to use heuristics early in order to reduce the time spent on the first DP phase, which is in contrast to IDP1.

In the multilevel optimization framework, shown below, we initially use a heuristic to identify a connected subgraph containing no more than k vertices (line 3). We then remove this subgraph and merge the vertices to obtain a new temporary relation $\mathcal{T}$. The subgraph that was removed is then optimized using a centralized optimization algorithm (line 6). k acts as an upper bound on the number of relations that the centralized optimization algorithm can optimize in a feasible time frame. We store the best plan obtained by this optimization in the optPlans structure for $\mathcal{T}$. This process is repeated until we arrive at a graph containing $\leq k$ vertices. In this case we have no need to apply the heuristic as the centralized algorithm is capable of optimizing the remaining graph. At this stage we have a number of optimized plans stored in opt-plan that may contain temporary relations. In the final phase we build the final plan by performing a series of temporary relation plan substitutions, where optimized plans P for temporary relations $\mathcal{T}$ are substitued in place of the existence of $\mathcal{T}$ in any optimal subplans. We term this class of algorithms *Multilevel* as the algorithms consist of a number of different levels, each of which involve a subset of relations being grouped together and optimized *completely*. IDP1ccp is *not* considered to be a Multilevel algorithm as at each phase we only perform a partial optimization while considering the complete set of relations.

Input: Connected join graph $G = (V, E)$ and the maximum optimization unit size k.
Output: An optimized query plan for G.

Algorithm ML-FRAMEWORK(G, k)
1 $opt\text{-}plans \leftarrow \emptyset$
2 **while** $|V| > k$ **do** $\{$
3 $S \leftarrow$ HEURISTIC(G, k)
4 **generate** new symbol: $\mathcal{T}$
5 $H \leftarrow$ MERGE-VERTICES$(G, S, \mathcal{T})$
6 $O \leftarrow$ OPTIMIZE(H)
7 $opt\text{-}plans(\mathcal{T}) \leftarrow O$
8 $\}$
9 **generate** new symbol: $\mathcal{J}$
10 $O \leftarrow$ OPTIMIZE(G)
11 $opt\text{-}plans(\mathcal{J}) \leftarrow O$
12 **return** BUILD-FINAL-PLAN($\mathcal{J}$)

The Multilevel Framework algorithm is shown above, which describes the general structure of the multilevel algorithms. The worst case running time and space complexity of the ML-framework algorithm are dependent on the time and space complexity of the heuristic and centralized optimization algorithm used.

6.2 SeqML : Sequential Multilevel Algorithm for Query Optimization

The sequential multilevel optimization algorithm (SeqML) proposed here adopts the ML framework as detailed above. The heuristic used in SeqML is the min-intermediate-result heuristic, which aims to reduce the size of intermediate results as done with the minimum selectivity heuristic proposed in [19]. The centralized optimization algorithm used is the DPccp algorithm as described in the previous section. Let $mat(P)$ denote the materialization site of plan P, i.e. the site at which plan P produces the results. The SeqML algorithm is given below.

Input: The query join graph $G = (V, E)$ and the maximum optimization unit size k.
Output: An optimized query plan for G.

Algorithm SEQ-ML(G, k)
1 $opt\text{-}plans \leftarrow \emptyset$
2 **while** $|V| > k$ **do** {
3 $S \leftarrow$ MIN-INTERMEDIATE-RESULT(G, k)
4 **generate** new symbol: $\mathcal{T}$
5 $H \leftarrow$ MERGE-VERTICES$(G, S, \mathcal{T})$
6 $O \leftarrow$ DPCCP(H)
7 $opt\text{-}plans(\mathcal{T}) \leftarrow O$
8 add $\mathcal{T}$ to catalog with resident site $mat(O)$.
9 }
10 **generate** new symbol: $\mathcal{J}$
11 $O \leftarrow$ DPCCP(G)
12 $opt\text{-}plans(\mathcal{J}) \leftarrow O$
13 **return** SEQ-BUILD-FINAL-PLAN$(\mathcal{J})$

The min-intermediate-result procedure is very similar to the minSel heuristic, discussed in section 2.3.2, in that it aims to reduce the size of intermediate results. The min-intermediate-result procedure operates as follows. We initially begin with two sets of vertices V and S where V is the vertex set of the join graph $G = (V, E)$ and S is empty. We first choose the edge (v_1, v_2) with the highest selectivity (lowest σ value) and add the vertices v_1 and v_2 to S. We then consider the vertices in the neighbourhood of S and add the vertex in $\mathcal{N}(S)$ such that the intermediate result obtained is the lowest possible. We repeat this process of picking a vertex from the neighbourhood that results in the lowest intermediate result until $|S| = k$. The min-intermediate-result procedure makes use of the intermediate-result procedure(S,v), which gives an estimate of the size of intermediate result produced when joining a plan involving relations in S and v. It does this by multiplying all existent edges (s, v) such that $s \in S$ and multiplying this conjunctive selectivity by the cardinalities of S and C. The min-intermediate-result algorithm is shown below.

Input: The query join graph $G = (V, E)$ and the maximum optimization unit size k.
Output: A subset of vertices that meets the minimum intermediate result heuristic.

Algorithm MIN-INTERMEDIATE-RESULT(G, k)
1 **find** $A, B \in V$ **such that**
  $\sigma(A, B) = min\{\, \sigma((a', b')) \mid (a', b') \in E\}$
2 $S \leftarrow \{A, B\}$
3 **while** $|S| < k$ **do** {
4 $N \leftarrow \mathcal{N}(S)$
5 **find** $C \in N$ **such that**
    INTERMEDIATE-RESULT(S,C) = $min\{$INTERMEDIATE-RESULT$(S, C') \mid C' \in N\}$
6 $S \leftarrow S \cup \{C\}$
7 }
8 **return** S

Line 8 of SeqML specifies that when we obtain an optimized plan O for a given connected subset S the subsequent temporary relation $\mathcal{T}$ is added to the catalog with resident site given as $mat(O)$. This implies that the final plan is built in a bottom up fashion, where the optimization results of earlier levels affect the results of subsequent levels.

The Seq-build-final-plan algorithm is shown below, which details how we obtain the final plan in terms of the original relations. We simply begin with the plan produced in the final level and continue to substitute optimized plans for temporary relations until the plan only consists of input relations. The *replace-child(parent,$\mathcal{T}$,P)* procedure is used to replace the child node $\mathcal{T}$ of node *parent* with the plan P.

Input: The symbol $\mathcal{J}$ corresponding to the optimized plan of the final level.
Output: The overall query execution plan in terms of the original relations.

Algorithm SEQ-BUILD-FINAL-PLAN$(\mathcal{J})$
1 $P \leftarrow opt\text{-}plans(\mathcal{J})$
2 **for all** nodes $n \in P$ **do** {
3 $P' \leftarrow opt\text{-}plans(n)$
4 **if** $P' \neq \emptyset$ **do** {
5 REPLACE-CHILD(PARENT(n), n, P')
6 }
7 }
8 **return** P

6.2.1 Worst Case Time Complexity

Let us derive the worst case time complexity of SeqML. Given a join graph containing n relation vertices we will iterate over $l = \lceil \frac{n}{k-1} \rceil$ many optimization levels. In the optimization levels we perform the min-intermediate result heuristic, which has a time complexity of $\mathcal{O}(n^2)$. We also use the merge procedure in each of the levels, where the time complexity of this procedure is $\mathcal{O}(k)$. The worse case time complexity of DPccp has been given previously (section 5) as $\mathcal{O}(s^3 \cdot 3^n)$ where s is the number of execution sites in the system. Since we perform DPccp on subgraphs containing at most k vertices this time complexity will contribute by $\mathcal{O}(s^3 \cdot 3^k)$ at each level. This implies that the worst case time complexity of SeqML can be given by the following.

$$
\begin{aligned}
&= \mathcal{O}(l \cdot (k + n^2 + s^3 \cdot 3^k) \\
&= \mathcal{O}(\lceil \frac{n}{k-1} \rceil \cdot (k + n^2 + s^3 \cdot 3^k)) \\
&= \mathcal{O}(\frac{n^3}{k} + \frac{n}{k} \cdot s^3 \cdot 3^k)
\end{aligned}
$$

6.2.2 Example : SeqML

Let us consider the join query in Figure 22 with the resident site information given in Figure 43. We assume that there are three execution sites in the system S_1, S_2 and S_3. Also, assume that the query site is S_1.

Relation	Resident Sites
R_1	S_1
R_2	S_1, S_2
R_3	S_2, S_3
R_4	S_2

Figure 43: Example Resident Site Information

Let the value of k be 2. In this situation the min-intermediate-result heuristic simply equates to choosing the edge with the lowest selectivity. In this case the chosen edge corresponds to the edge between R_1 and R_2. We extract the subgraph containing these relations (Figure 44) for optimization and merge them into the temporary relation $\mathcal{T}$ to obtain the graph shown in Figure 45.

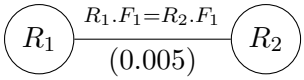


Figure 44: Extracted Join Graph: Level 1

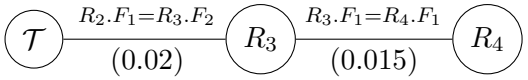


Figure 45: Join Graph: Level 2

After optimizating the join graph in Figure 44 it is likely that the best plan involves joining R_1 and R_2 at S_1 as this involves the least number of I/Os and no network communication. This plan is shown in Figure 46. The resident site of the temporary relation $\mathcal{T}$ is considered to be S_1 as this is the materialization site of the optimized plan. We then proceed as before by applying the min-intermediate-result heuristic to the join graph in Figure 44. This involves extracting the join graph containing the relations R_3 and R_4 as the edge between them has the minimum selectivity. The extracted graph can be seen in Figure 48 and the remaining graph can be seen in Figure 49 (with the new temporary relation $\mathcal{S}$). As R_3 and R_4 are both present at S_2 the optimal plan will consist of joining these relations at S_2. This plan can be seen in Figure 47.

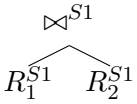


Figure 46: Optimized Plan from level 1

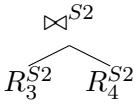


Figure 47: Optimized Plan from level 2

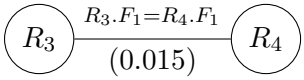


Figure 48: Extracted Join Graph: Level 2

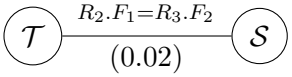


Figure 49: Join Graph: Level 3

Assume that the cardinalities of $\mathcal{T}$ and $\mathcal{S}$ are such that the final optimized plan will involve sending $\mathcal{S}$ to S_1 and performing the join between $\mathcal{T}$ and $\mathcal{S}$ at S_1. The optimized plan P is shown in Figure 50. The final plan is obtained by substituting previously optimized subplans into P and is given in Figure 51.

6.3 DistML : Distributed Multilevel Algorithm for Query Optimization

In this section we explore the possibility of distributing the optimization process so that we can make full use of the available system resources. The main idea of the distributed ML algorithm (Dist-ML) is to produce an optimization *task* on

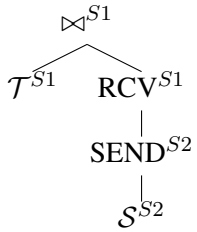


Figure 50: Optimized Plan for final level

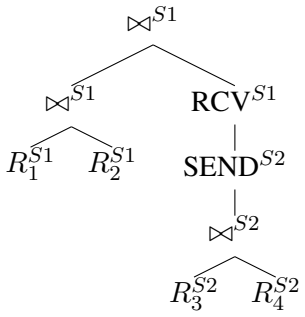


Figure 51: Final Plan

each level and send this task to an *optimization site*, which can then perform the optimization. Here we make the distinction between an *optimization site* and an *execution site*. An optimization site is used solely to optimize tasks. An execution site refers to sites that are capable of executing query operators. However, it is possible to consider a distributed system consisting of s sites where each site fulfills the role of an optimization site and an execution site.

The main advantage gained from distributing the optimization is that many optimization tasks may be able to be optimized concurrently by multiple different sites, thereby causing a linear speed up in optimization time given by the number of optimization sites. However, if we try to convert the sequential algorithm SeqML directly into a distributed algorithm we encounter the problem that the optimization in level i *may* depend on the result of the optimization in level $i-1$ (for $i > 1$). However, these dependencies do not always arise. Let $S_{Gi} = (V_i, E_i)$ denote the subgraph chosen for merging by the heuristic in level i. Let the temporary relation that replaces S_{Gi} be $\mathcal{T}_i$. When $\mathcal{T}_i \notin V_{i+1}$ we have no dependency between levels i and $i + 1$. The existence of dependencies means that it would not be possible to perform the optimizations concurrently at different sites. In SeqML we are required to wait for an optimization to finish so that we can allocate the resident site of the new corresponding temporary relation $\mathcal{T}$ to the materialization size of the produced plan P (lines 6-8 in SeqML). In DistML we hypothesize that an optimized plan P, involving the relations with a combined resident site set of RS, will have a materialization site s such that $s \in RS$. Here, RS is the union of the resident sites of each relation that is to be involved in the plan P. This means that the choice of the materialization site of P is delayed and is made by the optimization step in the *following* optimization level. Once an optimization task is submitted it is now possible to proceed *immediately* to the next optimization level using the resident site set RS for the temporary relation $\mathcal{T}$.

Let the optimization site to which the optimization task involving $\mathcal{T}$ is allocated be denoted by s_{opt}. Let RS denote the possible resident sites of $\mathcal{T}$. At s_{opt} we store the $|RS|$ optimal plans that materialize the result of $\mathcal{T}$ at each execution site $s \in RS$. When all optimization tasks for all levels are finished we then build the final plan as follows. We request the optimal plan for the final level with materialization site equal to the query site from the appropriate optimization site. We then take this plan and identify any containing temporary relations $\mathcal{T}$. These relations have execution site annotations determined by the optimization phase. We then request the plan P with $mat(P)$ equal to the chosen site for $\mathcal{T}$ from the optimization site responsible for the optimization of the temporary relation. In this sense the sites allocated to subplans corresponding to temporary relations are determined in a top-down fashion, as opposed to the bottom-up fashion used in SeqML. The algorithms Dist-ML and Dist-build-final-plan are given below.

Note that in the case where we are optimizing plans for a **centralized database system** SeqML and DistML produce the **same** plans (when given equal k values). Hence, in the event where the number of optimization tasks > 1 and the number of optimization sites > 1 theoretically we always have an advantage of using DistML over SeqML.

Input: The query join graph $G = (V, E)$ and the maximum optimization unit size k. The set of optimization site identifiers C.
Output: An optimized query plan for G.

Algorithm DIST-ML(G, k, C)
1 $optimized \leftarrow \emptyset$
2 $waiting\text{-}to\text{-}be\text{-}optimized \leftarrow \emptyset$
3 $plan\text{-}sites \leftarrow \emptyset$
4 **while** $|V| > k$ **do** {
5 $S \leftarrow$ MIN-INTERMEDIATE-RESULT(G, k)
6 **generate** new symbol: $\mathcal{T}$
7 $H \leftarrow$ MERGE-VERTICES$(G, S, \mathcal{T})$
8 SUBMIT$(H, \mathcal{T}, C)$
9 add $\mathcal{T}$ to catalog with resident sites $\cup_{v \in H} RS(v)$.
10 }
11 **generate** new symbol: $\mathcal{J}$
12 SUBMIT$(G, \mathcal{J}, C)$
13 **return** DIST-BUILD-FINAL-PLAN$(\mathcal{J})$

Input: The symbol $\mathcal{J}$ of the optimized plan of the final level.
Output: The overall query execution plan in terms of the original relations.

Algorithm DIST-BUILD-FINAL-PLAN($\mathcal{J}$)
1 **wait while** *waiting-to-be-optimized* $\neq \emptyset$
2 $c \leftarrow$ *plan-sites*($\mathcal{J}$)
3 $P \leftarrow c$.PLAN($\mathcal{J}$, *query-site*)
4 **for all** nodes $n \in P$ **do** {
5 **if** $n \in$ *optimized* **do** {
6 REPLACE-CHILD(PARENT(n), n, c.PLAN($n, n.site$))
7 }
8 }
9 **return** P

As with SeqML we use the min-intermediate-result heuristic and the DPccp centralized optimization algorithm. The DistML procedure makes use of the submit procedure, shown below, which is responsible for keeping track of available optimization sites and issuing optimization tasks to them. Here, c.DPccp indicates that the optimization is carried out using DPccp at the optimization site c.

Input: The query join graph $G = (V, E)$, a symbol $\mathcal{T}$ to be assigned to the optimized plan of G and the set of optimization site identifiers.
Purpose: Allocates the task of optimizing the join graph G to an optimization site and executes the optimization.

Algorithm SUBMIT($G, \mathcal{T}, C$)
1 *waiting-to-be-optimized* $\leftarrow$ *waiting-to-be-optimized* $\cup \{\mathcal{T}\}$
2 **choose** $c \in C$
3 *plan-sites*($\mathcal{T}$) $\leftarrow c$
4 c.DPCCP(G)
5 *waiting-to-be-optimized* $\leftarrow$ *waiting-to-be-optimized* $- \{\mathcal{T}\}$
6 *optimized* $\leftarrow$ *optimized* $\cup \{\mathcal{T}\}$

6.3.1 Worst Case Time Complexity

In the worst case DistML only has one optimization site available to it thereby making it equivalent to SeqML in terms of running time. This implies that DistML has the same worse case running time as SeqML of $\mathcal{O}(\frac{n^3}{k} + \frac{n}{k} \cdot s^3 \cdot 3^k)$.

6.3.2 Example : DistML

In the DistML example we encounter the same join graph levels as in the SeqML example. However, we do not wait for the $\mathcal{T}$ plan or $\mathcal{S}$ plan to be optimized so that we can assign the temporary relations a resident site equal to the materialization site of the plan. Instead, we assign the resident site (RS) of the temporary relations as follows.

$$RS(\mathcal{T}) = RS(R_1) \cup RS(R_2) = \{S_1, S_2\}$$
$$RS(\mathcal{S}) = RS(R_3) \cup RS(R_4) = \{S_2, S_3\}$$

We have the following optimization tasks to assign to optimization sites.

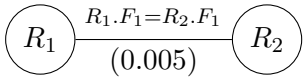


Figure 52: Optimization Task 1 : Level 1

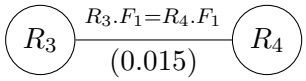


Figure 53: Optimization Task 2 : Level 2

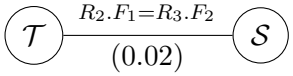


Figure 54: Optimization Task 3 : Final Level

In the event where the distributed system contains 3 or more optimization sites s_{opt} it is possible for these optimization tasks to be optimized concurrently thereby theoretically reducing the running time by a linear factor of s_{opt}. When an optimization site receives a task it saves optimized plans materialized at the union of all relation resident sites unless the task corresponds to the final level. In the case where an optimization site receives the final level task it only saves the plan that materializes the result at the query site. Assume that we have three optimization sites in the system. The plans stored at each of the three sites are shown in Figures 55-57.

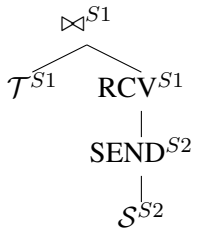


Figure 55: Plan stored at optimization site 1

Figure 56: Plans stored at optimization site 2

The final level plan, stored at optimization site 1, is materialized at S_1 as this is the query site. Let P denote a plan at optimization site 2 or 3 and let $RS(P)$ denote the union of all resident sites of all relations involved in P. Then $|RS(P)|$ plan versions are required to be stored at the optimization site, where each plan is materialized at each site in $RS(P)$. The tasks do not necessarily get optimized in any particular order. They are generated sequentially but it could be the case that a later task takes less time to complete than an earlier task. For example, it could actually be the case that optimization task 3 gets optimized before the other tasks. When all tasks are optimized we need to build the final plan. We begin with the plan from the top level (Figure 55) and request the plans that fit in place of the temporary relations from the optimization sites. Note that the site annotations of temporary relations must coincide between levels. The final plan can be seen in

Figure 57: Plans stored at optimization site 3

Figure 58. Note that in this case this plan corresponds to the plan obtained by the SeqML algorithm.

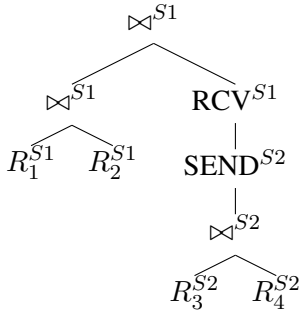


Figure 58: Final Plan

7 Implementation

In the previous section we discussed the novel multilevel optimization algorithms SeqML and DistML. In this section we give implementation details of the optimization algorithms SeqML and DistML. We also discuss the implementation of structures used by these classes such as join graphs, query execution plans and the system catalog. The implementation details for DPccp and IDP1ccp are not given here as the code is almost exactly as the algorithms given in section 5. The optimization algorithms discussed in this thesis were implemented in Java and the total code line count of the project was just above 10000. The project code can be found at the following URL.

http://www.comlab.ox.ac.uk/people/dan.olteanu/projects/distributedoptimization.html

7.1 Catalog

7.1.1 Internal Representation

As discussed in section 2, the system catalog is stored in the same way as all other data in a database system, i.e. in a table. In this thesis we focus on query optimization and do not consider implementing any part of the query executer. This means that the underlying mechanism to create, query and modify the system catalog is not available as we do not have a means of creating or manipulating actual relations. Instead of storing the catalog as a table we maintain the entire catalog in memory. This is sufficient for our current requirements as we do not consider systems with more than 100 relations and hence the catalog will not monopolize main memory.

The internal representation can be seen in Class Diagram 1. We can see that the Catalog class maintains several system wide variables that indicate the size of a disk page, the execution sites in the system and the time to transfer data across the network and to/from disk. The catalog also maintains a RelationEntry for each relation present in the system. The RelationEntry class maintains information about relations, such as their name, cardinality, tuple size etc. The sites at which the relation is available are also stored here along with the *Schema*. The Schema consists of a number of fields along with their corresponding DomainType. We fix the domain types to be one of four classes (A,B,C or D) where each type has their own size in bytes and number of different possible values.

```
<!DOCTYPE Catalog [
<!ELEMENT Catalog (CatalogEntry+)>
<!ELEMENT CatalogEntry (RelationEntry,FieldsEntry)>
<!ELEMENT RelationEntry (RelationName,NumberOfTuples,ResidentSites)>
<!ELEMENT FieldsEntry (DomainType,FieldName)>
<!ELEMENT ResidentSites (IPaddress+)>
<!ELEMENT IPaddress (#PCDATA)>
<!ELEMENT RelationName (#PCDATA)>
<!ELEMENT NumberOfTuples (#PCDATA)>
<!ELEMENT DomainType (#PCDATA)>
<!ELEMENT FieldName (#PCDATA)>
]>
```

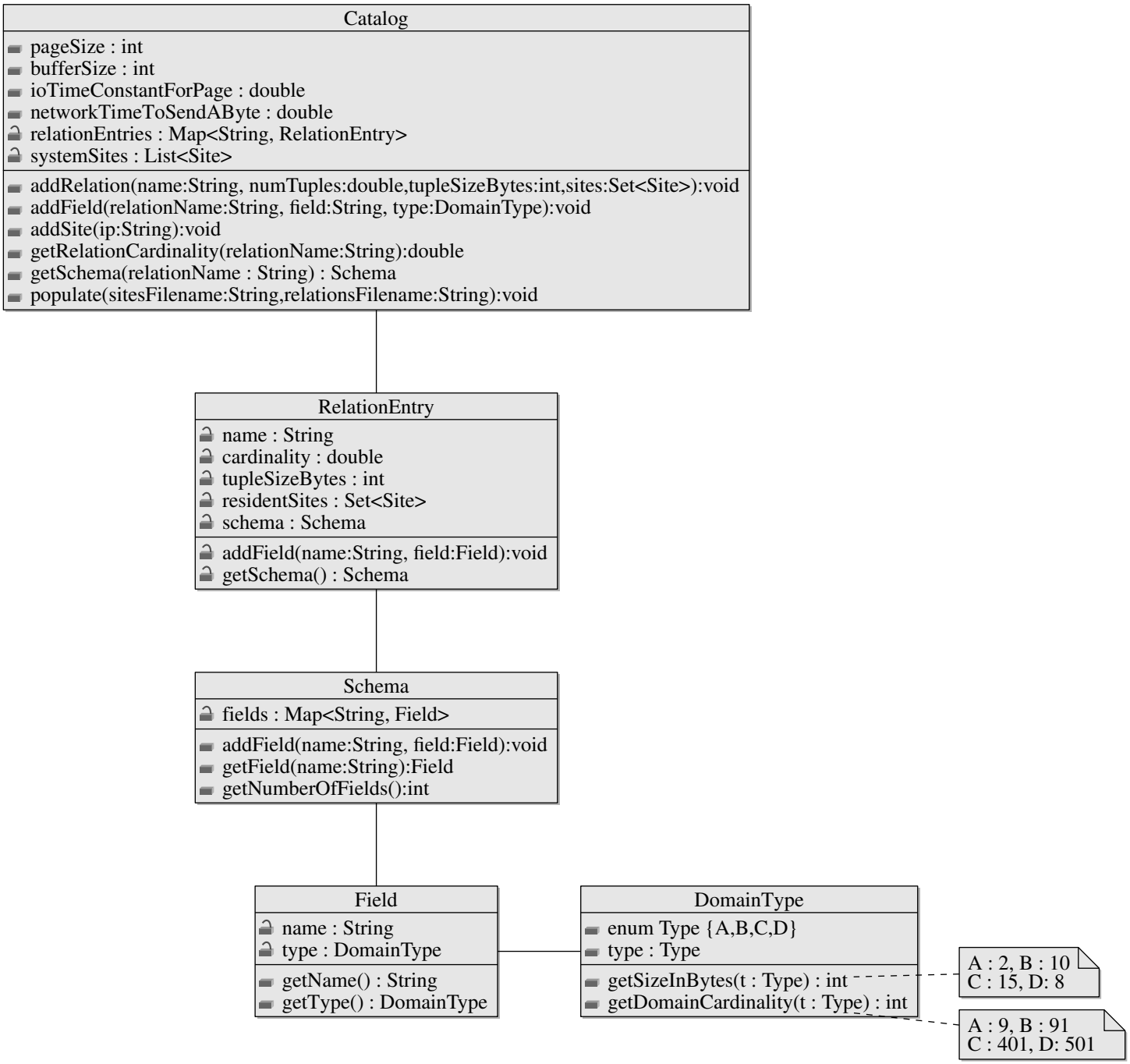
Figure 59: Catalog file format

7.1.2 File Format

We provide a flat file format for the catalog to allow catalogs to be saved to disk and later reused. A DTD indicating the format of this file is given in Figure 59 and a short catalog instance can be seen in XML format in Figure 60. An example of a compressed version of a catalog file used in the system can be seen in Appendix B.1. The file consists of a list of pairs of lines (l_1, l_2). l_1 contains a relation name, the cardinality, the size of each tuple in bytes and the list of resident sites. This corresponds to the RelationEntry element shown in Figure 59. We make the assumption that all relations have fixed length tuples, although the model could easily be adapted to allow for variable length tuples. The second line l_2 consists of a list of field entries, where each entry specifies the domain type of the field and the name of the field. This corresponds to the FieldsEntry element in Figure 59.

```
<Catalog>
    <CatalogEntry>
        <RelationEntry>
            <Name> A </Name>
            <NumberOfTuples> 1020 </NumberOfTuples>
            <TupleSize> 42 </TupleSize>
            <Resident Sites>
                <IP Address> 163.1.88.0 </IP Address>
                <IP Address> 163.1.88.1 </IP Address>
            </Resident Sites>
        </RelationEntry>
        <FieldsEntry>
            <DomainType> B </DomainType>
            <Name> A._F1 </Name>
        </FieldsEntry>
        <FieldsEntry>
            <DomainType> C </DomainType>
            <Name> A.F2 </Name>
        </FieldsEntry>
    </CatalogEntry>
</Catalog>
```

Figure 60: Short Example Catalog



Class-Diagram-1-:-System-Catalog

7.2 Join Graph

7.2.1 Internal Representation

Join graphs are used in our system to represent queries. The internal representation of a join graph consists of a JoinGraph class containing a single *graph* attribute of type Map<String, Map<String, JoinInfo>>. The String keys r_1 and r_2 correspond to the names of relations and the JoinInfo value encapsulates the join selectivity and join condition of the join between relations r_1 and r_2. When presented with the names of two relations we are able to find any join information present in constant time (assuming that the appropriate number of hash buckets is used and that there is a uniform distribution of keys). We are also able to check if a given edge is present in the join graph in constant time under these assumptions. However, this representation requires the JoinInfo entry to be stored twice, once for the edge (v_1, v_2) and once for the edge (v_2, v_1). Note that we only store the *reference* and not the object twice.

7.2.2 File format

In order to save a query to disk we provide a flat file representation of the query, which is almost identical to the internal representation. The first line of the file contains a list of space separated strings that correspond to the names of relations involved in the query. Each line l after the first line stores adjacency list of vertices for each vertex. The format of the file is shown by the DTD in Figure 61 and a short instance of this DTD can be seen in Figure 62. This is a very small example of a join graph containing two relations A and B that have a single join condition between them. A compressed version of a join graph file used in the system can be seen in Appendix B.2.

```
<!DOCTYPE JoinGraph [
<!ELEMENT JoinGraph (Relation+, AdjacencyList+)>
<!ELEMENT Relation (RelationName)>
<!ELEMENT AdjacencyList (VertexName, AdjacentVertex+)>
<!ELEMENT AdjacentVertex (VertexName,JoinCondition)>
<!ELEMENT JoinCondition (FieldName, FieldName, Selectivity)>
<!ELEMENT RelationName (#PCDATA)>
<!ELEMENT VertexName (#PCDATA)>
<!ELEMENT FieldName (#PCDATA)>
<!ELEMENT Selectivity (#PCDATA)>
<!ELEMENT FieldName (#PCDATA)>
]>
```

Figure 61: Join graph file format

7.3 Query Execution Plan

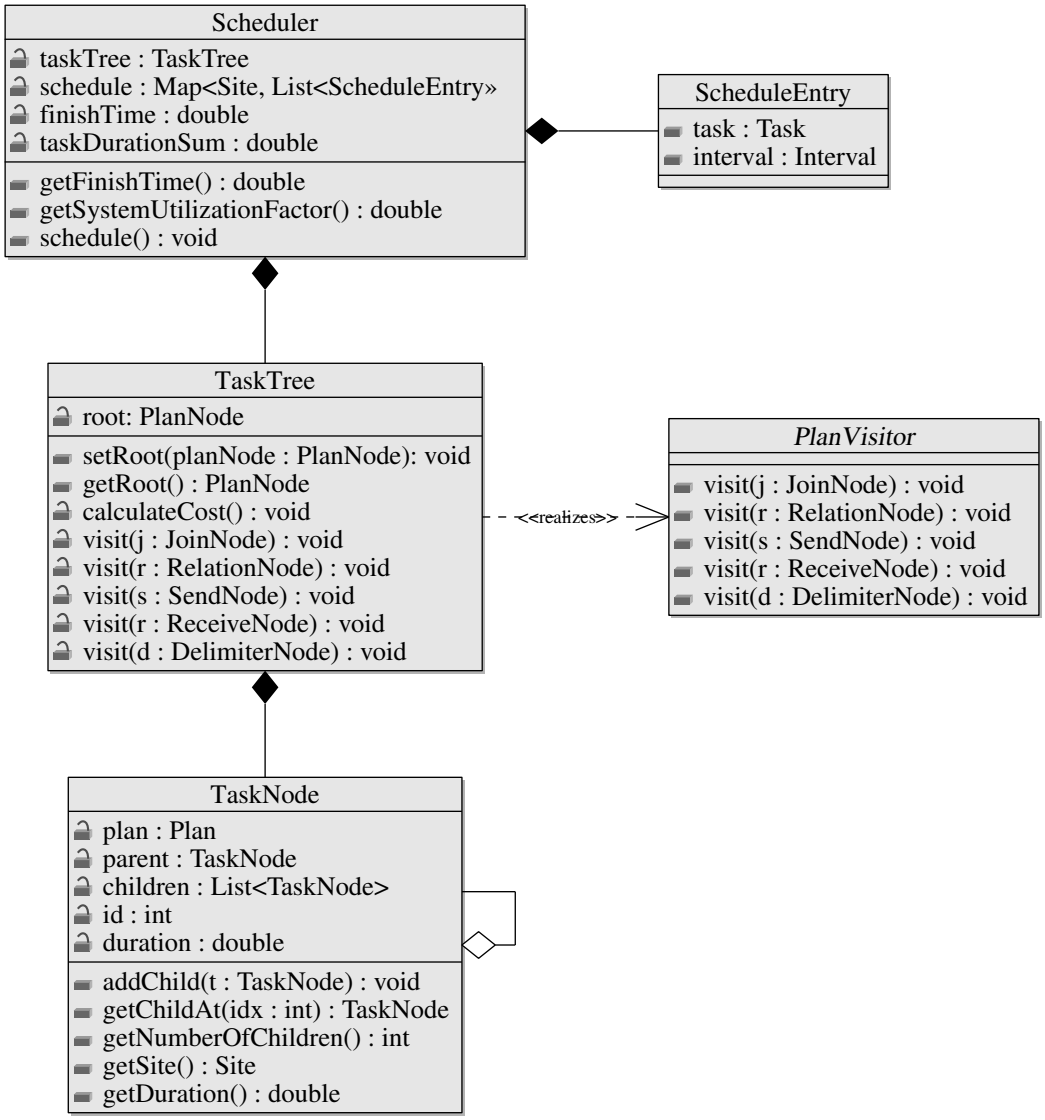
Class diagram 3 shows the design of the classes used to represent a Query Execution Plan. We adopt the composite design pattern to create the tree structure that represents the plan. The abstract class PlanNode is used to provide common functionality such as calculating output number of pages, total output bytes etc. It also enforces the interface regarding child manipulation to fit the composite pattern. The nodes that can be present in a query execution plan consist of JoinNode, RelationNode, SendNode, ReceiveNode and the DelimiterNode. As described in section 4 the purpose of the DelimiterNode is to isolate areas of a plan that belong to the same scheduling *task*.

The Plan class adopts the adapter design pattern by wrapping around the *root* PlanNode and providing methods to determine the cost of the plan. When the getCost() method is called on a Plan object a TaskTree object is created. During the initialization of a TaskTree the supplied Plan is cloned and the delimiter nodes are added to the appropriate locations within the cloned Plan to identify tasks. The cost of each task is then evaluated and stored in the task's duration field. The resulting task tree is then passed to a Scheduler, that performs the post order traversal of the tree and creates the final schedule. The main field of the Scheduler class is the *schedule* field, which has the type Map<Site, List<ScheduleEntry>>. Each site has an associated list of schedule entries, which are stored in this structure. A Schedule Entry simply consists of a task with an interval denoting when the task is scheduled to begin and finish. The classes involved in Scheduler component can

be seen in Class Diagram 2.

```
<JoinGraph>
    <Relation>
        <Name> A </Name>
    </Relation>
    <Relation>
        <Name> B </Name>
    </Relation>
    <AdjacencyList>
        <VertexName> A </VertexName>
        <AdjacentVertex>
            <VertexName> B </VertexName>
            <JoinCondition>
                <FieldName> A._F1</FieldName>
                <FieldName> B.F2</FieldName>
                <Selectivity> 0.01 </Selectivity>
            </JoinCondition>
        </AdjacentVertex>
    </AdjacencyList>
    <AdjacencyList>
        <VertexName> B </VertexName>
        <AdjacentVertex>
            <VertexName> A </VertexName>
            <JoinCondition>
                <FieldName> B.F2 </FieldName>
                <FieldName> A._F1 </FieldName>
                <Selectivity> 0.01 </Selectivity>
            </JoinCondition>
        </AdjacentVertex>
    </AdjacencyList>
< /JoinGraph>
```

Figure 62: Short Example Join graph



Class-Diagram-2-:-Scheduler-Component

Class-Diagram-3-:-Query-Execution-Plan

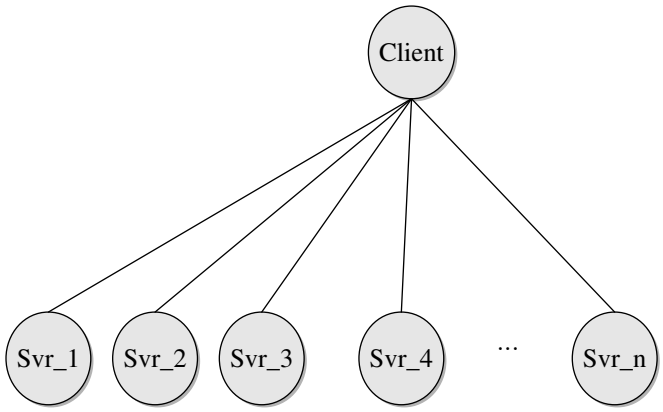


Figure 63: DistML Setup

7.4 Multilevel Optimization Code

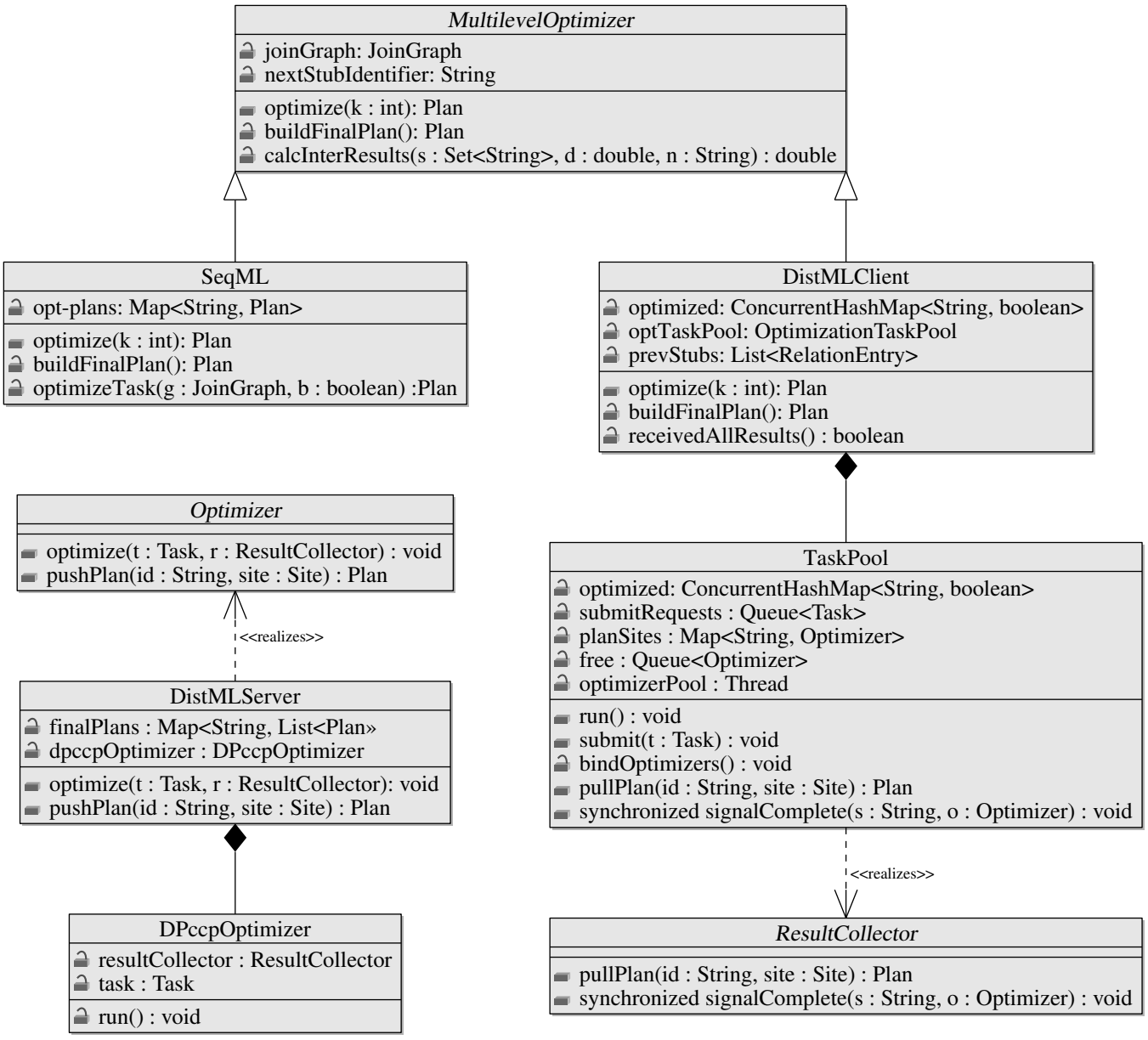
Class Diagram 4 shows the structure of the multilevel optimization component. The abstract class MultilevelOptimizer enforces that SeqML and DistMLClient implement the *optimize* and *buildFinalPlan* methods. It also provides an implementation of the common method *calcInterResults* that implements the min-intermediate-result heuristic. The SeqML class provides the implementation of the sequential multilevel algorithm and the DistMLClient class provides the implementation that coordinates the running of the distributed multilevel algorithm. The SeqML class provides the protected optimizeTask method that applies DPccp to the given subgraph. This method takes a flag, which indicates to DPccp whether or not the query site should be taken into account when optimizing the query. In the final level of the ML algorithms we require that the result is materialized at the query site but in intermediate levels this is not the case. When the flag is set to true DPccp chooses the plan in its final phase in a greedy manner, whereas when the flag is set to false the plan that is materialized at the query site is chosen.

In order to execute the distributed multilevel algorithm we need to create the required overlay network structure containing a single client (DistMLClient) and n optimization sites (DistMLServer) that take on the role of servers (shown in Figure 63). The client site is the site that received the query and wishes to execute the DistML algorithm. The overlay network is created using Java Remote Method Invocation (RMI) as it provides a simple means to program using multiple sites in a network without burdening the programmer with network programming details. Each optimization site executes an instance of the DistMLServer class,

which implements the Optimizer interface. RMI allocates a global identifer to the Optimizer object running on each server. This identifier is a URL of the form 'rmi://ipAddress:p/handle', where handle is a name given to the object whose reference is available on port p at IP address *ipAddress*. In this case we use the handle *OptimizerServer* at each optimization site.

When the DistMLClient code is run the optimization task produced at each level of the algorithm is passed into the TaskPool object. The TaskPool stores a queue of tasks that are waiting to be optimized as well as a queue of optimization sites that are currently not being used to optimize a task. The main responsibility of the task pool object is to allocate tasks to optimization sites that are free while considering load balancing. We do not wish to overload a small number of servers with optimization task requests as this may increase the likelihood of a server failure. Using a queue of available sites helps to avoid the overloading of servers.

DistMLServers receive a reference to the task pool so that they can indicate to it when they have completed optimizing their task. The TaskPool can then mark the temporary result corresponding to the task as completed in the *optimized* structure. This structure has a type ConcurrentHashMap<String, boolean>, which maps the name of the temporary result to a flag indicating whether or not it has been optimized. Note that it is necessary to use a *concurrent* hash map as many optimization sites may indicate to the task pool that they have completed their task at the same time. When the DistMLClient has issued all optimization tasks to the TaskPool it then waits until all tasks have been optimized. It can find this information by checking the ConcurrentHashMap structure to see if all tasks are mapped to 'true'. Once all tasks have been completed DistMLClient will build the final plan. This involves *pulling* the optimized plans for the temporary relation $\mathcal{T}$ from the sites that optimized $\mathcal{T}$. The optimization sites that optimized a task corresponding to the temporary relation $\mathcal{T}$ are stored in the TaskPool object.



Class-Diagram-4-:-Multilevel-Algorithms

8 Experiments

8.1 Setup

All experiments were performed on identical machines with the following cpu information - 64-bit AMD Athlon(tm) Dual Core Processor 4600+. The operating system used was Fedora with kernal version-2.6.29.6. The available memory was 1.94GB.

8.2 Algorithms

The algorithms used in our experiments are the sequential multilevel algorithm, named SeqML (section 6.2), the distributed multilevel algorithm, named DistML (section 6.3), and the improved iterative dynamic programming algorithm, named IDP1ccp (section 5.3). In SeqML we are given a query join graph G and a value k. We apply the min-intermediate-result heuristic until we obtain a subgraph S_G that contains k relations. We then remove S_G and replace it with a single temporary relation $\mathcal{T}$ to produce a new join graph G' containing $k-1$ less relation vertices than before. S_G is then optimized using the centralized optimization algorithm DPccp (section 5.2) and the obtained plan is saved under the $\mathcal{T}$ entry. We repeat the process of applying the heuristic to obtain a subgraph S_G, replacing the S_G with a temporary relation and optimizing S_G until the join graph contains less than or equal to k relation vertices. When we reach this final level we apply DPccp to the join graph for a final time to obtain the optimized plan in terms of the temporary relations. From this plan we subtitute saved plan entries until we obtain the final plan in terms of the original relations.

DistML is very similar to SeqML in the sense that they both apply the min-intermediate-result heuristic and optimize a subgraph at each level. The main difference is that DistML makes use of a number of optimization sites that perform the DPccp optimizations possibly concurrently. To allow this concurrent processing we have had to adopt a heuristic when indicating to upper levels where temporary relations are materialized. We take the union of all sites of relations involved in a subgraph and indicate that $\mathcal{T}$ is a resident at all of these sites. This is in constract to SeqML where we wait for a subgraph to be optimized and then specify that the corresponding temporary relation $\mathcal{T}$ is resident at the materialization site of the optimized plan. In the worst case the optimized plan of an upper level will disagree with the location of $\mathcal{T}$ with the level directly below, which will cause an additional ship node to be added to the plan. This can be the cause of optimality compromises

in the overall plan. Given the same value of k DistML and SeqML will produce the same plan when the system contains only one execution site, i.e. in a centralized database system.

IDP1ccp takes as input the join graph G of a query and a parameter k. Let the balanced value of k be denoted by b (see section 5.3). We perform the first dynamic programming phase by generating access plans, 2-way plans, ..., b-way plans. We then apply a greedy heuristic and choose a b-way plan P with the lowest cost. Any plan saved in memory that involves any relation of P is removed from memory and we denote the plan P by $\mathcal{T}$. We then calculate the new balanced value based on k and the number of remaining relations waiting to be involved in the plan. We continue to apply the DP phase and the greedy heuristic until we arrive at the final plan involving $\leq k$ temporary relations. We then substitute the temporary relation plans into the last obtained plan to produce the final plan in terms of base relations. For large queries and large k values the time required for the first DP phase can be debilitating.

8.3 Query Generator

In order to perform experiments involving IDP1ccp and the ML algorithms it is necessary for us to be able to generate a set of queries in join graph form. However, we cannot produce queries without having relations to refer to. We therefore have to generate a system catalog before generating queries. Consider the experimental case where we have a distributed setting consisting of s_e execution sites and where the maximum number of relations involved in a query is n. The minimum number of relation entries required to be present in the catalog must therefore be n. We build the catalog by randomly generating a name for a relation and then randomly assigning it a cardinality given by the probabilities in Table 2.

Probability	Cardinality
5%	1000-10000
40%	10000-100000
25%	100000-1000000
30%	1000000-10000000

Table 2: Relation Cardinality Probabilities

The next step is to populate the relation entry with fields. Each relation entry is assigned between 5 and 10 fields according to a uniform probability distribution. As with [12], we assign each field a domain according to the probabilities in Table

3. The size of each domain and the size in bytes can also be seen in Table 3. The final addition to each relation entry is the resident sites of the relation. We first choose randomly how many sites the relation should be available at using 1+U where U is a discrete random variable taking values from 0 to $s_e - 1$. We then proceed by picking the required number of sites at random and allocating them to the given relation entry.

Domain Type	Probability	Domain Size	Bytes
A	5%	9	2
B	50%	91	10
C	30%	401	15
D	15%	501	8

Table 3: Field Domain Probabilities

A *query generator* has been implemented to randomly generate queries of a particular join graph structure and of a particular size. The query generator is capable of generating queries of any size and of the type chain, cycle, clique, star or mixed. To create a mixed join graph we could randomly place join conditions between relations thereby creating arbitrary join graphs. This is likely to produce queries that would not be representative of real life queries. Instead, we make mixed join graphs by fusing component join graph structures (chain, cycle, clique and star) together as follows. We require a mixed join graph containing n relations. We randomly choose a value between 1 and n inclusive, say x. We then randomly choose a join graph component structure, where each structure has an equal chance of 0.25 of being chosen. A join graph of the structure conforming with the chosen component containing x relations is then randomly created. We then have $n - x$ relations left to include in the join graph G. We repeat the process to produce another component c. A join edge is then added between a random vertex in G and a random vertex in c. We repeat these steps until a join graph containing n relations is created. Class diagram 5 in Appendix D shows the query generator component classes.

Once a join structure is generated we then need to produce a completed join graph by adding join information onto edges. Our initial approach to randomly generating join conditions between two relations R_1 and R_2 was to randomly pick two fields $R_1.Fi$ and $R_2.Fj$ for equality and calculate the selectivity as follows. Let dom(Fi) denote the number of possible unique values of the domain type of attribute Fi.

$$\sigma(R_1, R_2) = \frac{1}{max(dom(Fi), dom(Fj))}$$

This led to the same problem as in [12] where intermediate results would commonly contain very small fractions of a tuple, which would be scaled up by subsequent joins. In practise we do not encounter intermediate results with fractions of tuples. To overcome this problem and to obtain more realistic queries we adopt the same solution as in [12], which is to randomly embed chains of key-foreign key pair join conditions within join graphs. We assign join information by visiting each vertex in order of decreasing degree and decreasing relation size within classes of the same degree. When at a vertex we consider each neighbour and assign the join condition randomly with probability 0.1 or as a key-foreign key pair with probability 0.9.

8.4 Design

The aim of our experiments is to determine the effectiveness of the ML algorithms and to identify situations in which it is more appropriate to apply an ML algorithm over IDP1ccp and vice versa. In order to be able to compare the algorithms we propose an optimization time cap of 30 seconds, which all runs of algorithms must comply with. The maximum k values that allow each algorithm to run in under 30 seconds are shown in Appendix F. We experiment with queries involving 20, 40, 60, 80 and 100 relations in distributed settings containing 1, 3 and 9 execution sites. In the case of DistML we use 10 optimization sites. We experiment with queries consisting of chain, cycle, star and clique join graphs. Mixed join graphs are not considered as determining appropriate k values for the ML algorithms is a non trivial task and is left as further work.

Let us consider the setting where we have s_e execution sites, a join graph of type t and queries containing n relations. We have 20 randomly generated queries for each of the possible settings arising from picking $s_e \in \{1, 3, 9\}$, $t \in \{Chain, Cycle, Star, Clique\}$ and $n \in \{20, 40, 60, 80, 100\}$. In the first experiment we aim to discover the best value of the paramater k to use for each algorithm for each of the possible settings just described. The best k value for a possible setting $\mathcal{S}$ will be the value that results in the lowest averaged scaled cost of the 20 plans for a particular setting. This means that we optimize each of the 20 queries for all k values up to the maximum value. For a given query we scale the costs of plans obtained for each k value so that the minimum plan cost is 1. We then average all scaled results

for each of the 20 queries to produce the averaged scaled cost of each k value for all 20 queries. This will give us an indication if there is a k value that results in a universally low plan cost (for our 20 randomly generate queries).

In experiment 2 we use the k values discovered from experiment 1 to compare the IDP1ccp, DistML and SeqML algorithms by the optimality of the plans they produce. We again use averaging of 20 queries to present the scaled cost of plans produced by the algorithms for a particular setting. The main aim of experiment 2 is to identify situations in which is it benefical to use a particular optimization algorithm over another one.

8.5 Experiment 1: Quality of ML Algorithms with Varying k

In this section we present the experimental results for experiment 1, where we wish to determine the value of k that results in the ML algorithms returning the plan with the lowest cost. Let x-query denote a query involving x relations. The graphs on the following page show the SeqML averaged scaled cost for the 80-query and 100-query for the different query types (chain, cycle, star or clique).

8.5.1 Chain and Cycle Analysis

The first observation to make is that in all chain and cycle graphs there exists values of k that result in the ML algorithms producing plans that are orders of magnitude more costly than the optimal plan. Also, note that there exist values of k that give plans with a very low cost. A possible reason for the peaks in cost is as follows. Let us consider the number of optimization levels l and the number of relations n_f involved in the final level of the ML algorithms. The number of levels can be given by the following equation, where n is the number of relations in the query and k is the size of subqueries.

$$l = \lceil \frac{n}{k-1} \rceil \tag{8}$$

The size of the final level n_f is given by the following equation.

$$n_f = n - (l-1)(k-1) \tag{9}$$

Graph 1 : SeqML - Chain 80-Query, 3 Execution Sites

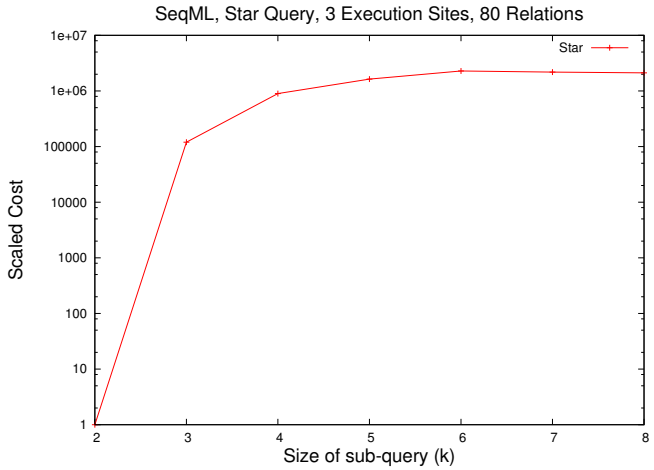
Graph 2 : SeqML - Chain 100-Query, 3 Execution Sites

Graph 3 : SeqML - Cycle 80-Query, 3 Execution Sites

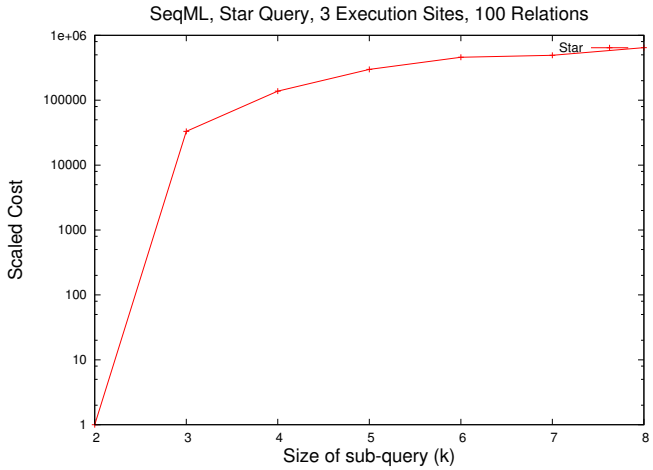
Graph 4 : SeqML - Cycle 100-Query, 3 Execution Sites

Graph 5 : SeqML - Clique 80-Query, 3 Execution Sites

Graph 6 : SeqML - Clique 100-Query, 3 Execution Sites



Graph 7 : SeqML - Star 80-Query, 3 Execution Sites



Graph 8 : SeqML - Star 100-Query, 3 Execution Sites

In some events where the final level is restricted to containing only a small number of relations/temporary relations the optimizer has little choice but to choose a poor plan. For example, consider the case where we have a distributed setting with 3 execution sites and where we wish to optimize a chain query involving 100 relations. In Graph 2 we can see a large spike in cost where $k = 7$. When $k = 7$ we perform 17 levels and have a final level containing just 4 relations. With each level in the algorithm we perform a heurstic, which can be the cause of plan quality degradation. The small size of the final level imposes a restriction on the optimizer, which reduces the optimization options available. Similarly when $n = 80$ k values of 7 and 8 lead the final level to containing just 2 and 3 temporary relations respectively. This again accounts for the spikes at $k = 7$ and $k = 8$ in the Graph 1. Cycle queries also exhibit this behaviour. An example, of this can be seen in Graph 4 for $k = 15$. Using equation 9 we can see that when $k = 15$ the final level contains just 2 relations, which can cause a very poor plan to be produced.

Let us now turn our attention to the case of finding *promising* values of k. We wish to be able to reduce the number of levels, while restricting the optimizer as little as possible in the final level. Consider the 100-query chain case. Some values of k that give a large final level are 14 and 16. In Graph 2 we can see that $k = 14$ actually gives the plan with the lowest cost for all k values from 2-18. However, $k = 16$ does not give such a good plan. This is an indication that the ML algorithms do not possess the property of graceful degradation, where the performance of the algorithm declines monotonically with a decrease in k. However, by choosing some large enough k values we can obtain very good plans. An example of this can be seen particularly in the case where we consider a cycle 100-Query in a system containing a single execution site. When we make k large enough to just

Figure 64: Cycle 100-Query, SeqML in a centralized system

past 50 we start to obtain some very good plans as shown in Figure 64.

Note that in a distributed setting consisting of 3 execution sites k values up to 18 were tested for chain queries while k values only up to 16 were tested for cycle queries. This is as a result of obeying the optimization time cap of 30 seconds.

8.5.2 Clique Analysis

It is clear from looking at Graph 5 and Graph 6 that the ML algorithms arrive at better plans as k increases. When the optimized query execution plans were inspected it was found that the intermediate results quickly became very small, which is due to the high number of join conditions available. Consider the second optimization level in the case where we have an 80-Query with $k = k'$. The temporary relation $\mathcal{T}$ present in the second level will contain k' relations and the number of relations total in level 2 will be $80 - k' + 1$. Let the edge e be the edge between $\mathcal{T}$ and any other relation r in the second level. Since the query is of a clique form e will have k' conjunctive join conditions. The selectivity of this overall join edge is calculated by multiplying selectivities of these conjunctive join conditions therefore as k increases the join condition becomes more selective. For larger values of k the optimizer has more flexibility to find the joins that will result in producing a low intermediate result very quickly. Figure 65 shows the schedule of the plan produced for the clique 20-query when $k = 2$. Note that

some of the tasks are large (e.g. 1, 5, 9) and consist of a set of sequential joins at a single site. With a larger k value of 7 not only do we obtain a join condition that is more selective quickly but as we can see from Figure 66 the optimizer has more flexibility to take advantage of inter-operator parallelism. The best choice of k for clique queries is the largest possible in accordance with the optimization time cap.

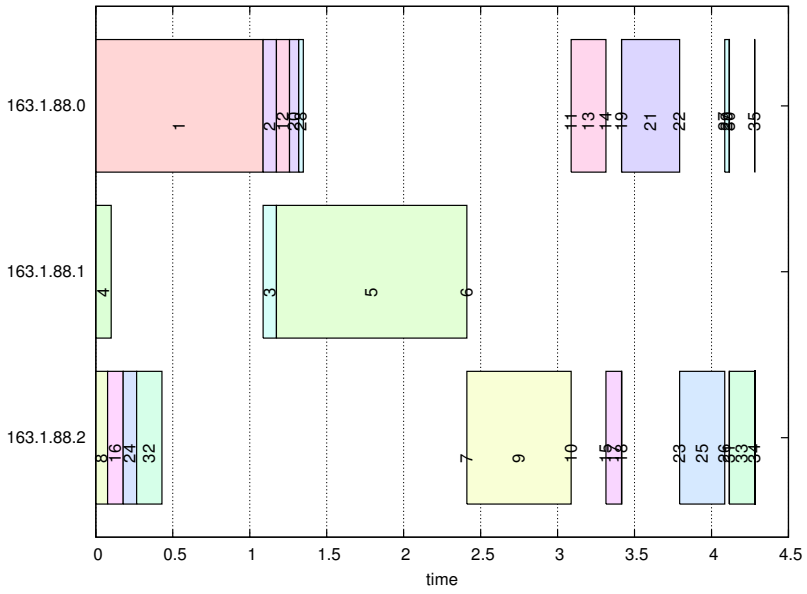


Figure 65: 20-Query Clique Schedule, k=2

8.5.3 Star Analysis

It can be seen from Graph 7 and Graph 8 that as k increases the ML algorithms produce increasingly worse plans. Since all relations in a star query must join with the centre relation the plans for join queries are sequential in nature. By using the min-intermediate-result heuristic the intermediate results build up with the number of relations added into the query plan. By increasing k we produce better and better subplans that keep intermediate results low at the early stage of the plan. However, by using the min-intermediate-result heuristic we effectively save the worse for last, where the intermediate results are large and hence where it makes the biggest impact on the cost. Let p_2 be the plan obtained for the star 80-query with $k = 2$ and let p_7 denote the plan obtained when $k = 7$. We noticed that subplans rooted

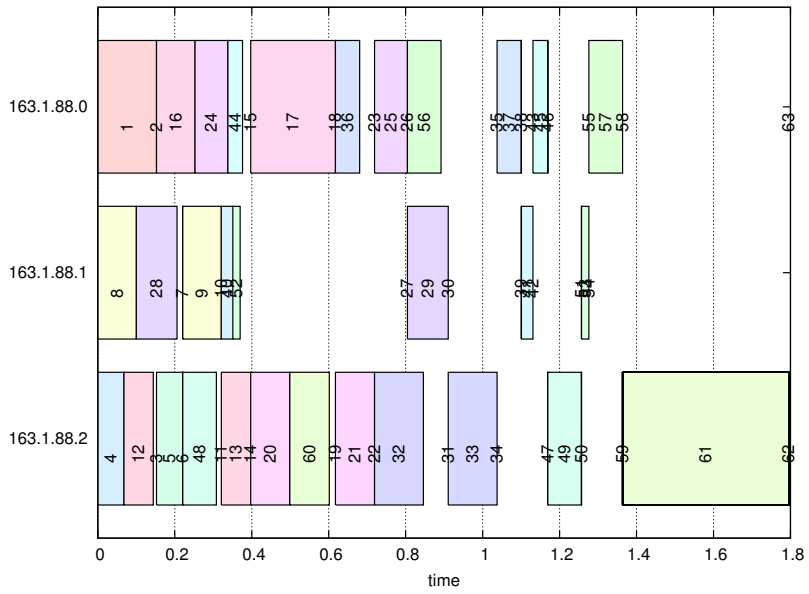


Figure 66: 20-Query Clique Schedule, k=7

at about 5 joins lower down from the root contained lower intermediate results for p_7. However, the cost of the remaining 5 or so joins became very expensive due to the low remaining selectivities and large cardinality of base relations involved. When $k = 2$ the final 5 or so joins were less expensive as some small relations still remained due to being overlooked by the min-intermediate-result heuristic. Recall that we choose the *first* edge according to selectivity information alone and we do not consider the cardinality of adjacent relations in this initial choice. We only consider cardinalities after the first choice. We can therefore deduce that the min-intermediate-result heuristic is not well suited to star queries and we should choose $k = 2$ for further experiments involving star queries with the ML algorithms.

8.6 Experiment 2 - Algorithm Comparison

In this section we present the results from our second experiment, which compares the scaled optimality of the plans found by SeqML (Red in graphs), DistML (Green) and IDP1ccp (Blue). The best plan found by any algorithm is given the scaled cost of 1 and the other plans produced by the remaining algorithms have their cost scaled in accordance with the best plan. We present the results when

considering a distributed system consisting of three execution sites. The results when considering a distributed system containing one execution site and nine execution sites can be found in Appendix E.

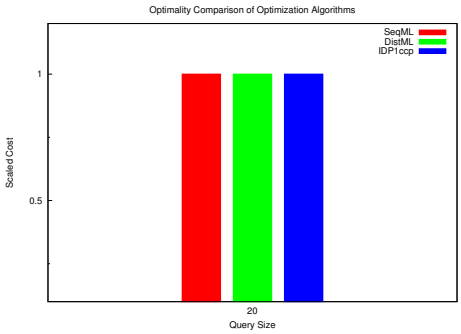
8.6.1 Chain Analysis

The comparison graphs for chain queries can be seen in Graphs 9-13. In graph 9 we can see that all algorithms produce the same plan. In this case all algorithms use DPccp because of the small size of the query (20 relations), i.e. the ML algorithms only have a single level and IDP1ccp only has a single DP phase. Graph 10 shows the optimization for 40-Query. Clearly we can see that IDP1ccp produces a plan that is orders of magnitude better than the ML algorithms. This may indicate that the min-intermediate-result heuristic is not as successful as IDP1ccp in identifying relations that should be grouped together to produce subplans of the overall query.

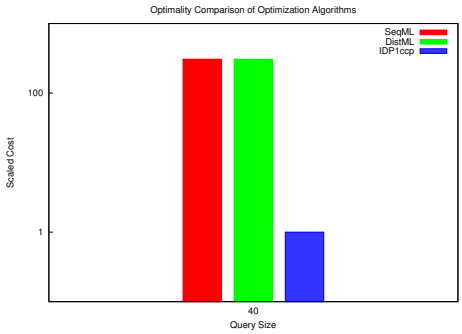
In Graph 11 we can see that the algorithms produce plans that have roughly the same cost. SeqML narrowly has the smallest cost and IDP1ccp narrowly has the highest cost. By observing Graphs 11-13 we can see that IDP1ccp's performance degrades significantly when compared with the ML algorithms. This is due to it being forced to use smaller and smaller k values, to comply with the time cap, as the size of the query being optimized increases. When optimizing 20-Query IDP1ccp used a k value of 20 but when optimizing 60-Query it is reduced to using $k = 9$. As shown in the experimental results in [9] the quality of IDP1 degrades as k decreases.

In Graph 12, where 80-Query is being optimizer, we can see that IDP1ccp now produces the plan with the worst cost (about 5 times worse than the plan produced by DistML). The cost of the plan produced by SeqML is about 4 times worse than DistML. Again, the reason for the degradation in the performance of SeqML is due to the lower k value. SeqML is only able to use k values up to 18, while DistML can use values up to 25. SeqML choses to use $k = 17$ as this results in 5 levels of optimization and will produce a final level containing 16 (temporary) relations thereby giving the optimizer a significant amount of freedom in the final level. As DistML has access to more resources, i.e. 10 optimization sites, it will choose a k value of 21 as this results in only 4 optimization levels and a final level containing 20 (temporary) relations. DistML allows the optimizer to have more freedom in the final level and has less levels, which contributes to it producing better results than SeqML. This trend continues into Graph 13, where IDP1ccp produces a plan with a cost that is orders of magnitude worse than the plan produced by DistML.

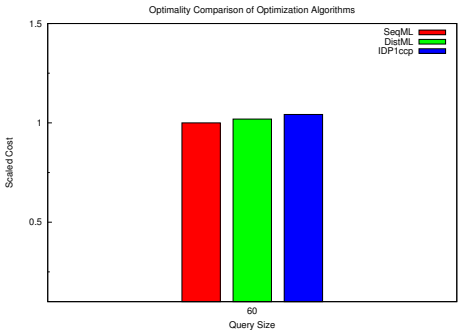
SeqML similarly produces a very poor plan as a result of a restriction on k due to the time cap.



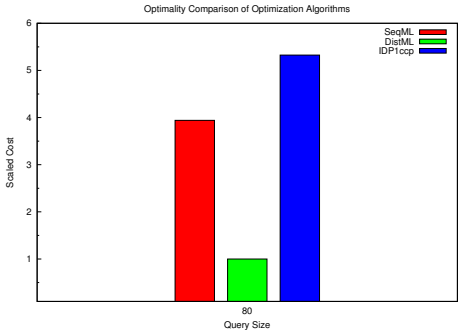
Graph 9 : Chain, 20-Query



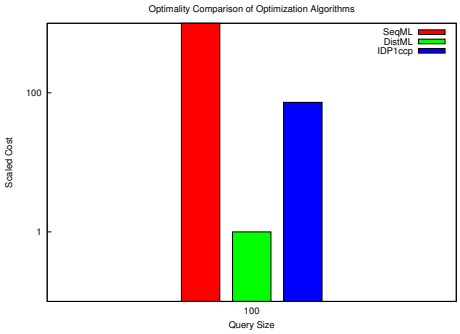
Graph 10 : Chain, 40-Query



Graph 11 : Chain, 60-Query



Graph 12 : Chain, 80-Query



Graph 13 : Chain, 100-Query

8.6.2 Cycle Analysis

The comparison graphs for cycle queries can be seen in Graphs 14-18. The outcome shown in Graph 14 is the same as in Graph 9 where all algorithms equate to DPccp due to the small size of query and small number of execution sites. In Graph 15 we can see that IDP1ccp produces the best plan while DistML and SeqML produce plans with cost factors of 22 and 134 respectively (note the log scale). The k value chosen by SeqML is 15 as it results in 3 optimization levels and produces a final level containing 12 (temporary) relations. This k value results in the maximum final level size possible. However, if we look at the average scaled cost of the 20 queries with varying k (Figure 67) we can see that a better k value would have been 17.

Figure 67: Scaled average cost of 20 queries involving 40 relations

When observing the results of varying k with 40-Query we found that the majority of graphs had a similar area around $k = 15$ as shown in Figure 68. However, 1 out of the 20 queries had a huge spike at $k = 15$ that has dominated the average. This graph is shown in Figure 69. This indicates that even though 15 is a promising value for k it does not always give plans with very good costs for all possible cycle queries involving 40 relations. The reason for this may be that the ML algorithm incorrectly groups some relations together using the min-intermediate-result heuristic in a level that results in a particularly expensive subplan, which becomes detrimental to the final plan.

We can see a similar situation in Graphs 16 and 17. In Graph 17 DistML performs particularly poorly by producing plans that are over 40 times worse than IDP1ccp.

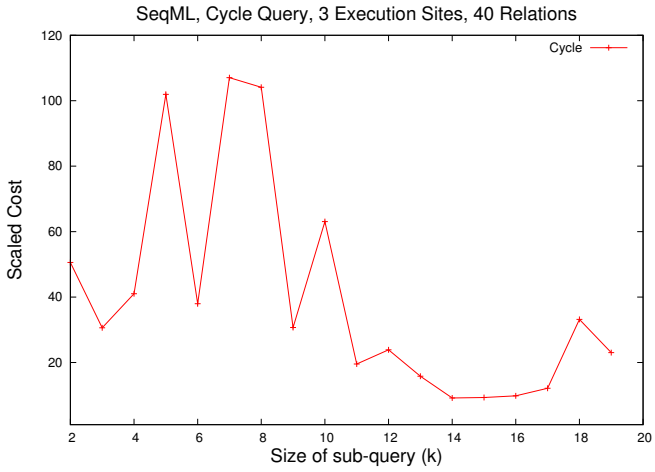


Figure 68: Scaled cost of plans for a single query (Query 8) with varying k

After investigating the cause of this we discovered that out of the 20 queries 8 queries resulted in comparable plan costs, 7 queries resulted in DistML producing better plans than IDP1ccp and 5 queries resulted in DistML producing worse plans than IDP1ccp. The main problem was that in one of the cases where DistML produced a worse plan than IDP1ccp the cost of this plan was over 1000 times that of the corresponding IDP1ccp plan, which has had a significant impact on the average cost. This would indicate that in a small number of cases the heuristic used by DistML to overcome the site dependency between optimization levels can induce plans whose cost is considerably far from the cost of the optimal plan.

In Graph 18 we can see that IDP1ccp has produced a plan that is more than 8 times worse than the plan produced by DistML. IDP1ccp has gone from using $k = 20$ for 20-Query to $k = 5$ for 100-Query, which is cause of the performance degradation. SeqML produces a plan with a worse cost than DistML for a similar reason.

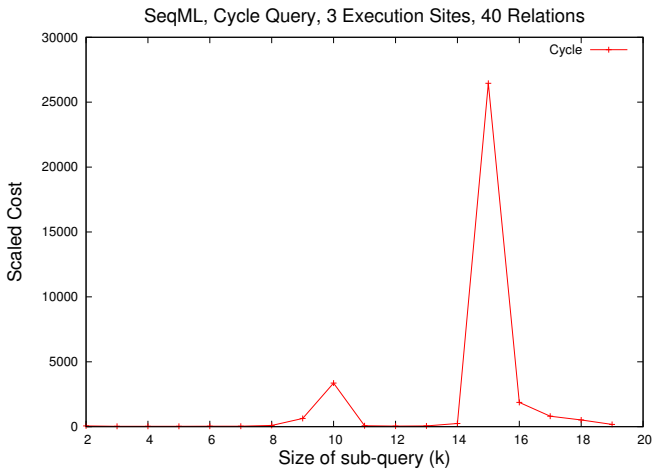
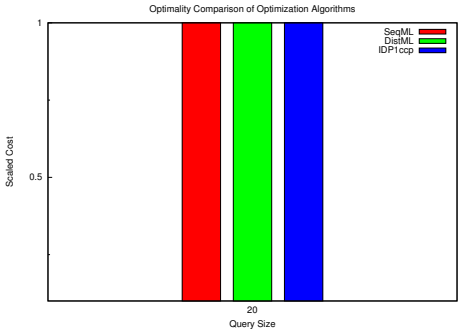
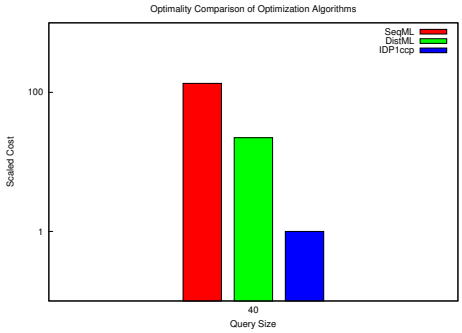


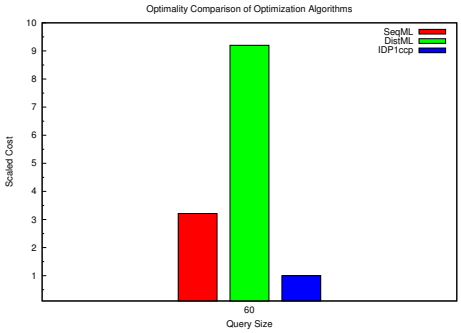
Figure 69: Scaled cost of plans for a single query (Query 20) with varying k



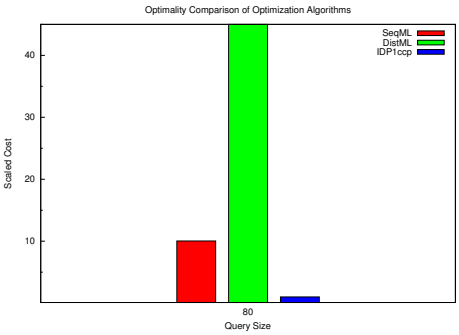
Graph 14 : Cycle, 20-Query



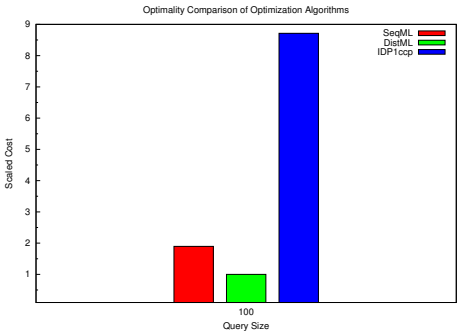
Graph 15 : Cycle, 40-Query



Graph 16 : Cycle, 60-Query



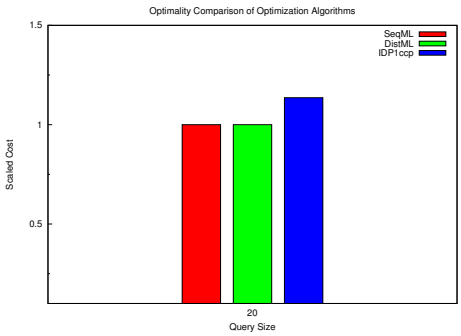
Graph 17 : Cycle, 80-Query



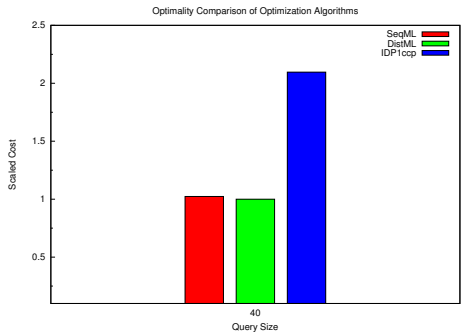
Graph 18 : Cycle, 100-Query

8.6.3 Clique Analysis

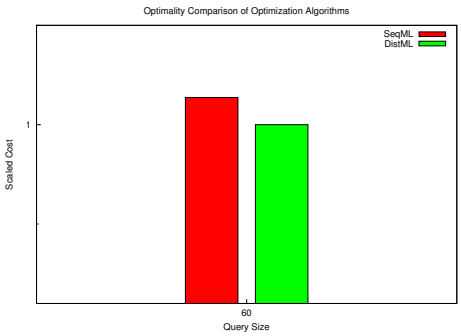
Graphs 19-24 show the scaled plan costs of applying the ML algorithms and IDP1ccp to clique queries. The first point to observe is that for queries involving ≥ 60 relations IDP1ccp could not finish within the time cap even for the minimim k value of 2. We only show the results for the ML algorithms in these cases. The complexity of IDP1ccp is such that only the k value of 2 could be used for queries involving less than 60 relations. By using $k = 2$ we reduce the opportunities of inter-operator parallelism and have a considerable number of DP phases where we use a greedy heuristic. This contributed to IDP1ccp producing the plan with the worst cost in the case of 20-Query and 40-Query. In all cases SeqML and DistML have very similar results, where in some cases DistML produces plans with slightly better cost. This again is due to DistML being able to use slightly larger values of k.



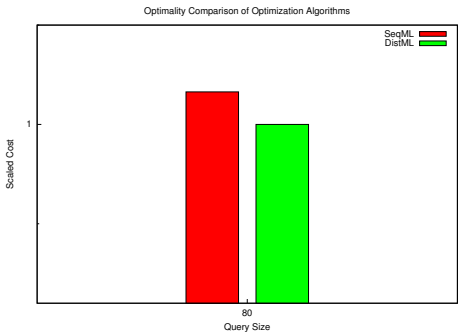
Graph 19 : Clique, 20-Query



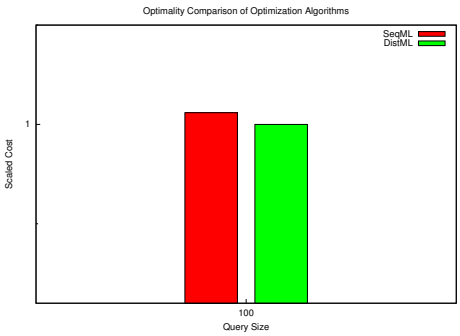
Graph 20 : Clique, 40-Query



Graph 21 : Clique, 60-Query



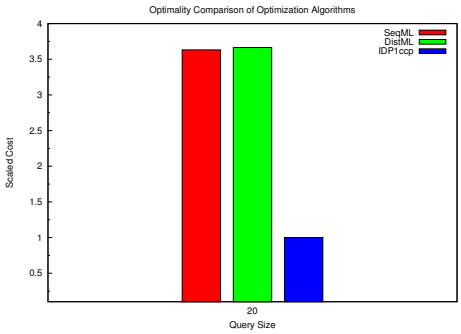
Graph 22 : Clique, 80-Query



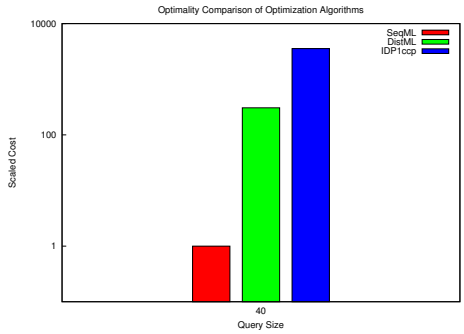
Graph 23 : Clique, 100-Query

8.6.4 Star Analysis

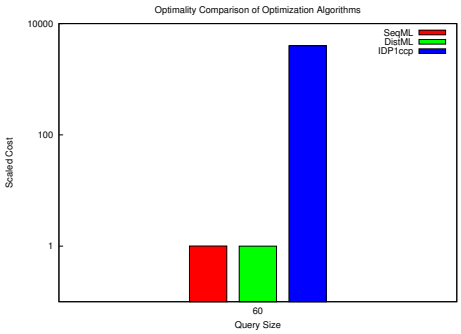
Graphs 24-28 show the algorithm comparisons for the experiments involving star queries. When optimizing 20-Query (Graph 24) IDP1ccp produces the best plan, with SeqML and DistML producing plans with a cost about 3.5 times higher. In Graph 25 we can see that IDP1ccp has produced the plan with a cost that is orders of magnitude greater than that produced by SeqML. Similarly, DistML has also produced a plan that has a cost that is orders of magnitude greater than the plan produced by SeqML. As with the cycle 80-Query case some queries produced by DistML were extremely poor due to a bad site choice for some temporary relations, which caused the average cost to be extremely poor. However, this is not the case for 60-Query, 80-Query and 100-Query where SeqML and DistML produce very similar results, while IDP1ccp produces plans with a very high cost. In Graph 28 IDP1ccp produces a plan with a cost that is over 10000 times worse than the plans produced by the ML algorithms.



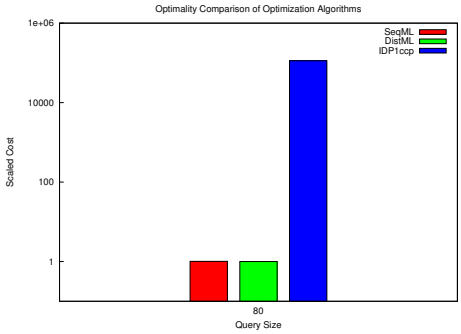
Graph 24 : Star, 20-Query



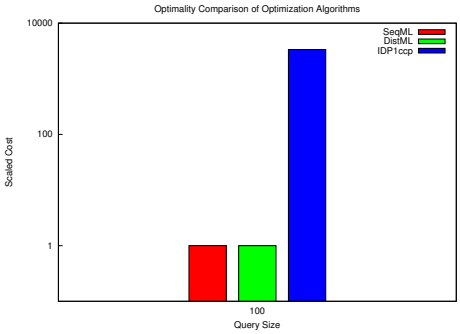
Graph 25 : Star, 40-Query



Graph 26 : Star, 60-Query



Graph 27 : Star, 80-Query



Graph 28 : Star, 100-Query

8.7 Summary

From experiment 1 we can see that the ML algorithms do not exhibit graceful performance degradation when optimizing chain and cycle queries. This means that as the optimization problem becomes more complex, i.e. the queries increase in size, the performance of the ML algorithms do not decrease monotonically. However, for large enough values of k we can produce plans that have a very low cost. For example, it was found that when considering a centralized database setting k values of just above 50 resulted in the ML algorithms producing very good plans when optimizing chain and cycle 100-Query. In general, we can identify possible promising k values for chain and cycle queries by considering values of k that lead to the final level involving large numbers of relations and by aiming to reduce the number of levels. The most extreme case of this of course is when we optimize the entire query in a single level, which would be the case were we have the minimum number of levels and the maximum final level size. This would result in the optimal result being produced if it was feasible to do so.

The best values of k to use for the ML algorithms for clique queries are the highest possible given the available optimization time cap. This gives the optimizer greater flexibility at early levels, which can result in very small intermediate results being obtained very quickly. However, star queries are quite the opposite. By using the min-intermediate-result heuristic with star queries with high k values we inadvertently leave joins involving larger relations with less selective join conditions until last. In star queries all relations have to join with the central relation, which means that the plan is sequential in nature where intermediate results get larger and larger (generally) as we traverse from the leaves to the root of the plan. By leaving larger relations to be joined last we produce very expensive final joins, which is detrimental to the entire cost of the plan. The very small intermediate results obtained in clique query plans due to the high number of conjunctive join conditions prevents this problem from occurring with clique queries.

In general, experiment 2 seems to indicate that both ML algorithms and IDP1ccp have strengths in different situations. IDP1ccp seems to use a heuristic that, when combined with phases of DP, produce better results for smaller queries than the ML algorithms. However, when IDP1ccp becomes unable to cope with larger queries or a distributed setting containing 3+ sites the ML algorithms produce significantly better plans. For example, in the case where we have nine execution sites (Appendix E.1.4), IDP1ccp is only able to produce plans for chain 20-40 queries and cycle 20-Query, while the ML algorithms are capable of producing plans for all sized queries for all query types. However, the k values used by the ML algorithms

are pushed down in larger distributed settings, which causes the quality of the plans produced to degrade.

In some cases the concurrent execution provided by DistML allows higher k values to be used, which can result in DistML obtaining better plans than SeqML. Examples of this can be seen in Graph 13 and Graph 18 where DistML produces considerably better plans than SeqML for the 100-Query chain and cycle queries. In the case of the chain query DistML is able to use a k value up to 24, while SeqML can only use a value up to 17. DistML chooses to use a k value of 21 as this produces a final level with 20 relations. If we look at the Appendix F we can see that this value of k corresponds to DistML obtaining particularly good plans but due to the time cap it is out of reach for SeqML.

9 Conclusions and Future Work

Determining the cost of a query execution plan (QEP) and searching through the vast space of possible equivalent plans for a given user query makes the task of query optimization in a distributed setting very challenging. The problem of determining the minimum possible response time of a query can be addressed by scheduling subcomponents of a QEP. This allows opportunities for inter-operator parallelism to be identified thereby resulting in a more accurate estimate of cost being obtained for a plan. In order for this schedule to be upheld a query executer would have to be adapted to include scheduling signals. This would involve a site notifying other sites when it is $x\%$ through a particular task so that other sites could begin their tasks at roughly the correct scheduled time. When dealing with very large relations the overhead of scheduling signals would become negligible.

Future work on the cost model used in this thesis would be required to produce a cost model that is capable of handling pipelining between operators, i.e. where one operator feeds its output tuples directly into a parent operator when they become available without writing them to disk. In this case we primarily mean pipelining between operators executing at different execution sites. It is likely that pipelining in a distributed system would cause multiple sites to be tied up, which would lead to less sites being available to take advantage of inter-operator parallelism. If we extended the system to allow fragments of relations to be replicated at multiple sites in the system then it would be necessary to extend the cost model further to handle intra-operator parallelism. Additional further work would involve determining the impact of using the improved cost model presented in this thesis against using cost models used in previous papers that are based only on resource consumption. It is likely that the improvement in quality of the plans produced by the cost model proposed in this thesis, particularly in large distributed systems, would be significant.

The size of the search space of equivalent plans has the impact that the performance of current enumeration algorithms begins to degrade significantly even for moderate sized systems containing about 8-10 execution sites. We found that IDP1ccp could barely optimize any queries in the given optimization time cap in a system containing nine execution sites. For smaller distributed systems, it was found that IDP1ccp is suitable for optimizing small to moderate sized chain and cycle queries (containing between 0 and 60 relations). However, for larger queries the quality of IDP1ccp degraded significantly as the size of the k parameter was reduced. In these situations the ML algorithms, particularly DistML, produced much better quality plans than IDP1ccp. In some situations DistML was able to produce much better

plans than SeqML. This was due to DistML being able to use larger k values as a result of utilizing multiple optimization sites concurrently. This provides an indication that in order to optimize large queries it is necessary to use the available resources in the system and distribute the optimization process and not just the execution process.

It was found that the ML algorithms occasionally produced plans that were significantly worse than the optimal plan. This is likely to be caused by the min-intermediate-result heuristic occasionally grouping relations together into subqueries that are unsuitable. In general, having many levels and small final levels was identified as being the cause of some very poor plans being produced. Further work could include finding a heuristic that causes the performance of the ML algorithms to degrade gracefully.

In general if we wish to be able to optimize plans in a distributed system it is advisable to use IDP1ccp for smaller queries, expecially for chain and cycle queries. When the size of queries becomes large or the distributed setting contains between 3-9 sites the ML algorithms should be used as they out performed IDP1ccp in this setting. There is still much work to be done to produce a distributed optimization algorithm capable of optimizing very large queries in large distributed systems. However, we believe that the ML algorithm framework and cost model presentated in this thesis provide a stepping stone in the right direction to achieve this goal.

References

[1] A. G. Arun Swami. Optimization of large join queries. In *ACM SIGMOD Conf Management of Data, Chicago, Ill*, pages 8–17, 1988.

[2] K. Bennett, M. C. Ferris, and Y. E. Ioannidis. A genetic algorithm for database query optimization. In *In Proceedings of the fourth International Conference on Genetic Algorithms*, pages 400–407. Morgan Kaufmann Publishers, 1991.

[3] C.-T. Y. Chih Yen, S. Tseng. Scheduling of precedence constrained tasks on multiprocessor systems. In *Algorithms and Architectures for Parallel Processing, ICAPP 95*, pages 379–382, 1995.

[4] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. In *SIGMOD Conference*, pages 9–18, 1992.

[5] Y. E. Ioannidis. Query optimization. In *ACM Computing Surveys*, pages 103–114, 1996.

[6] K. W. R. James F. Kurose. *Computer Networking: A Top-Down Approach Featuring the Internet*. Addison Wesley, 2009.

[7] H. L. Kian-Lee Tan. On resource scheduling of multi-join queries in parallel database systems. *Information Processing Letters*, pages 189–195, 1993.

[8] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32:422–469, 2000.

[9] D. Kossmann and K. Stocker. Iterative dynamic programming: A new class of query optimization algorithms. *ACM Transactions on Database Systems*, 25, 2000.

[10] O. P. L. R. Graham, D. E. Knuth. *Concrete Mathematics: A Foundation for Computer Science, 2nd ed.* Addison Wesley, 1994.

[11] C. S. Mullins. Tuning db2 sql access paths, Jan 2003. http://www.ibm.com/developerworks/data/library/techarticle/0301mullins/0301mullins.html.

[12] T. Neumann. Query simplification: Graceful degradation for join-order optimization. In C. Binning and B. Dageville, editors, *SIGMOD-PODS09 : Compilation Proceedings of the International Conference on Management of Data 28th Symposium on Principles of Database Systems*, pages 403–414, Providence, USA, June 2009. Association for Computing Machinery (ACM), ACM.

[13] D. Olteanu. Query optimization, Hilary Term 2009-2010. http://www.comlab.ox.ac.uk/teaching/materials09-10/databasesystemsimplementation/dsi11.ppt.

[14] K. Ono and G. M. Lohman. Measuring the complexity of join enumeration in query optimization. In D. McLeod, R. Sacks-Davis, and H.-J. Schek, editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 314–325. Morgan Kaufmann, 1990.

[15] F. P. Palermo. A database search problem. *Information Systems COINS IV, J.T. Tou, Ed, Plenum Press, New York, NY*, pages 67–101, 1974.

[16] G. Ramakrishnan. *Database Management Systems, Third Edition*. McGraw-Hill, 2003.

[17] W. Scheufele, G. Moerkotte, and S. A. Constructing optimal bushy processing trees for join queries is np-hard (extended abstract). Technical report, Universität, Mannheim, 1996.

[18] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, I. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *ACM SIGMOD International Conference on Management of Data.*, pages 23–34, 1979.

[19] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. *VLDB Journal*, 6:191–208, 1997.

[20] A. Swami. Optimization of large join queries: Combining heuristics and combinatorial techniques. In *ACM SIGMOD Conference on Management of Data, Portland, OR*, pages 367–376, 1989.

[21] G. M. Thomas Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *Proceedings of the 32nd international conference on Very large data bases*, pages 930–941. VLDB Endowment, 2006.

[22] B. Vance and D. Maier. Rapid bushy join-order optimization with cartesian products. In *In Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 35–46, 1996.

[23] E. W. Yannis E. Ioannidis. Query optimization by simulated annealing. In *ACM SIGMOD Conf Management of Data, San Francisco, Calif*, pages 9–22, 1987.

[24] Y. C. K. Yannis E. Ioannidis. Randomized algorithms for optimizing large join queries. In *ACM SIGMOD Conf Management of Data, Atlantic City, NJ*, pages 312–321, 1990.

[25] C. Z. Yingying Tao, Qiang Zhu and W. Lau. Optimizing large star-schema queries with snowflakes via heuristic-based query rewriting. In *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*, pages 279–293, Toronto, Canada, 2003. IBM Press.

Appendix

A IDP1ccp Component Algorithms

The main alterations made to the DPccp utility algorithms involve restricting the size of connected subsets produced. This is done by specifying the maximum size of connected subsets allowed using a parameter k. In enumerate-csg-rec the recursion step is halted if the size of the current connected subset is equal to the maximum allowed size-1. Similarly in enumerate-cmp we only produce complement connected subsets whose size combined with the given subset s is equal to the maximum allowed size.

Input:	Connected join graph G consisting of vertices $V = \{v_1, v_2, ..., v_n\}$ and edges E. A is used to store connected subgraphs of G. k is the maximum block size.
Precondition:	Vertices in G are numbered in a breadth first manner.
Output:	Obtains all subsets of V that form a connected subgraph of G. such that $\|V\| \leq k$.

Algorithm ENUMERATE-CSG'(G, A, k)

```
1  for i ← n down to 1 do {
2     A ← A ∪ {{v_i}}
3     ENUMERATE-CSG-REC'(G, {v_i}, B_i, A, k)
4  }
```

Input: Connected join graph G consisting of vertices $V = \{v_1, v_2, ..., v_n\}$ and edges E. S is a set of vertices that contain the current subgraph state. X is a set of vertices that cannot be explored further from subgraph S. A is used to store connected subgraphs of G. k is the maximum block size.
Precondition: Vertices in G are numbered in a breadth first manner.
Output: Recursively obtains all subsets of V that form a connected subgraph of G such that $|V| \leq k$.

Algorithm ENUMERATE-CSG-REC'(G, S, X, A, k)

```
1  N ← 𝒩(S) − X
2  for all S′ ⊆ N, S′ ≠ ∅, |S′| ≤ min(k − |S|, |N|) enumerate subsets first {
3     A ← A ∪ {S ∪ S′}
4  }
5  for all S′ ⊆ N, S′ ≠ ∅, |S′| ≤ min(k − |S|, |N|) enumerate subsets first {
6     ENUMERATE-CSG-REC'(G, S ∪ S′, X ∪ N, A, k)
7  }
```

Input: Connected join graph G consisting of vertices $V = \{v_1, v_2, ..., v_n\}$ and edges E. S is a connected subgraph of G. A is used to store results. k indicates maximum block size.
Precondition: Vertices in G are numbered in a breadth first manner.
Output: Obtains all connected subgraphs H such that (S, H) is a csg-cmp-pair and where $|S| + |H| = k$.

Algorithm ENUMERATE-CMP'(G, S, A, k)

```
1  X ← ℬ_min(S) ∪ S
2  N ← 𝒩(S) − X
3  for all v_i ∈ N by descending i {
4     A ← A ∪ {{v_i}}
5     ENUMERATE-CSG-REC'(G,{v_i}, X ∪ (ℬ_i ∩ N), A, k − |S|)
6  }
```

B Cost Model Example

B.1 Catalog

A 28937 88 163.1.88.0
B _A.F1 A A.F2 D A.F3 C A.F4 C A.F5 D A.F6 B A.F7 B A.F8 B A.F9
B 92882 91 163.1.88.0
B _B.F1 C B.F2 C B.F3 D B.F4 B B.F5 B B.F6 D B.F7 C B.F8
C 77574 80 163.1.88.1
C _C.F1 D C.F2 B C.F3 B C.F4 B C.F5 C C.F6 D C.F7 B C.F8
D 25000 105 163.1.88.1
C _D.F1 B D.F2 B D.F3 C D.F4 C D.F5 B D.F6 C D.F7 C D.F8
E 68548 74 163.1.88.2
D _E.F1 D E.F2 B E.F3 B E.F4 B E.F5 D E.F6 B E.F7 B E.F8
F 40000 70 163.1.88.2
C _F.F1 C F.F2 B F.F3 C F.F4 C F.F5
G 5000 100 163.1.88.1
C _G.F1 B G.F2 G.F3

B.2 Query

A B C D E F G
A B A.F2=B.F2 0.000005
B A B.F2=A.F2 0.000005 C B.F3=C.F3 0.000001
C B C.F3=B.F3 0.000001 D C.F5=D.F4 0.000005
D C D.F4=C.F5 0.000005 E D.F6=E.F5 0.00001
E D E.F5=D.F6 0.00001 F E.F1=F.F1 0.00001
F E F.F1=E.F1 0.00001 G F.F2=G.F3 0.00001
G F G.F3=F.F2 0.00001

C Chain ccp formula proof

Theorem D1: For all integers $n > 1$

$$\sum_{i=1}^{n-1} i(i+1) = \frac{n^3 - n}{3}$$

Proof. We use proof by induction. Assume true for n. Then we have the following.

$$\sum_{i=1}^{n-1} i(i+1) = \frac{n^3 - n}{3}$$

$$\sum_{i=1}^{n-1} i(i+1) + n(n+1) = \frac{n^3 - n}{3} + \frac{3n(n+1)}{3}$$

$$\sum_{i=1}^{n-1} i(i+1) + n(n+1) = \frac{n^3 - n + 3n^2 + 3n}{3}$$

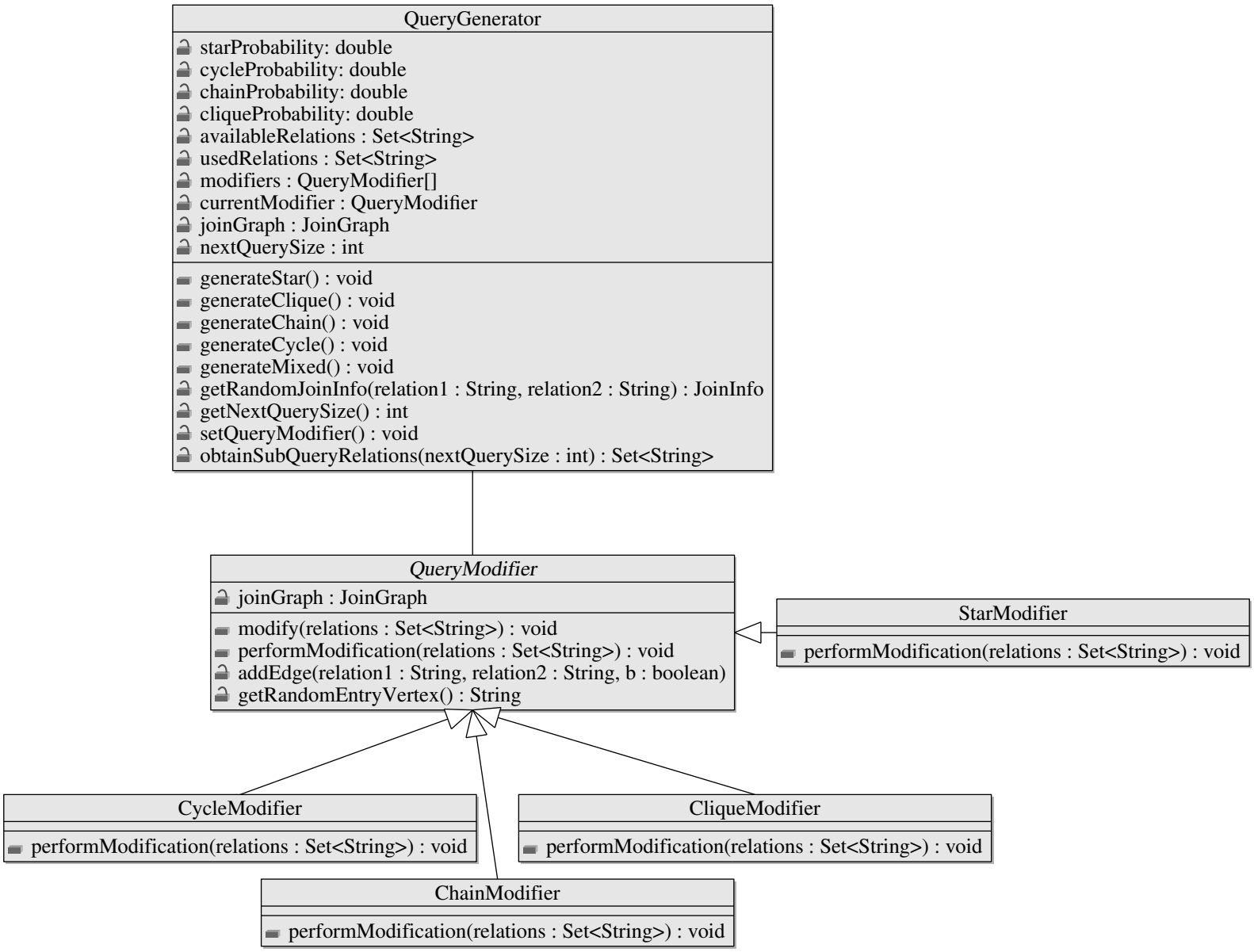
$$\sum_{i=1}^{n-1} i(i+1) + n(n+1) = \frac{(n+1)^3 - (n+1)}{3}$$

If D1 is true for n then it is true for $n+1$. When $n = 2$ lhs = 2 = rhs hence true for all $n > 1$. □

Theorem D2: The number of csg-cmp-pairs (#ccp) of a chain query involving n relations is $\frac{n^3-n}{3}$.

Proof. Let csg_i denote the set of connected subgraphs that contain i vertices.
For a chain graph we have $|csg_i| = n - i + 1$.
For a given $s \in csg_i$ there exists $(n - i)$ connected complement subgraphs.
The total number of ccps can be found by summing all ccp for each csg.
$\Rightarrow \#ccp = \sum_{i=1}^{n-1}(n - i + 1)(n - i)$
Let $j = n - i$.
$\Rightarrow \#ccp = \sum_{j=1}^{n-1} j(j+1)$
By theorem D1 we have the desired result. □

D Query Generator Design

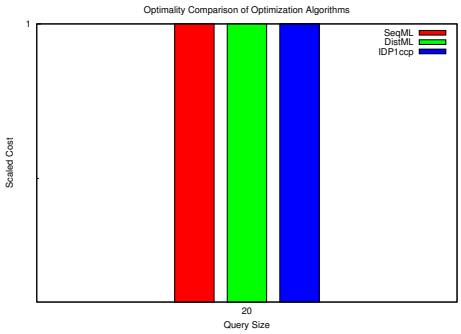


Class-Diagram-5-:-Query-Generator

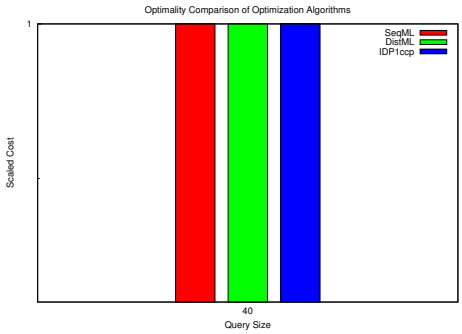
E Algorithm Experimental Comparison

E.1 Distributed database with one execution site

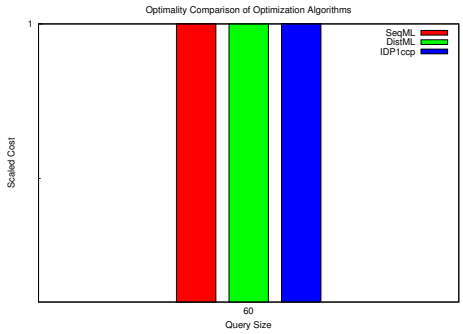
E.1.1 Chain Queries



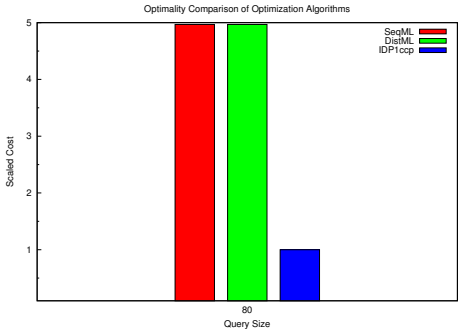
Graph 29 : Chain, 20-Query



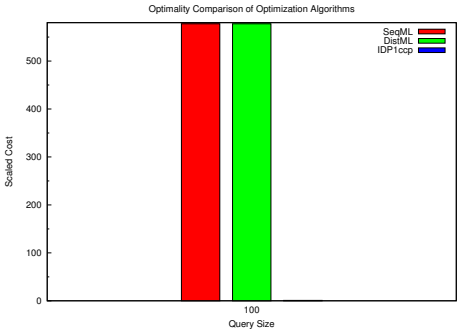
Graph 30 : Chain, 40-Query



Graph 31 : Chain, 60-Query

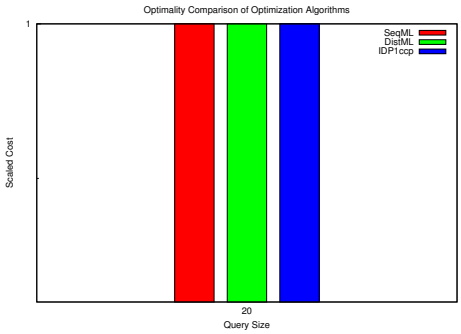


Graph 32 : Chain, 80-Query

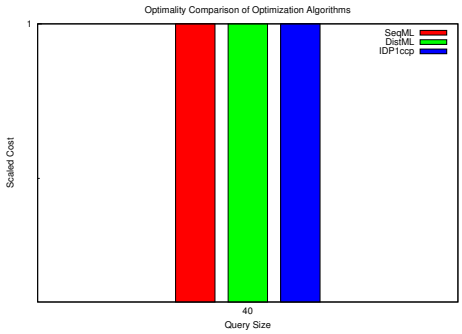


Graph 33 : Chain, 100-Query

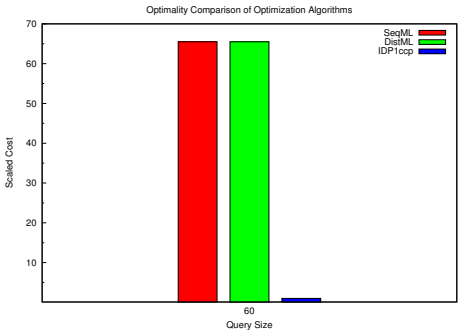
E.1.2 Cycle Queries



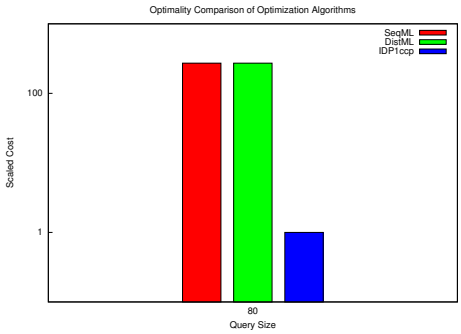
Graph 34 : Cycle, 20-Query



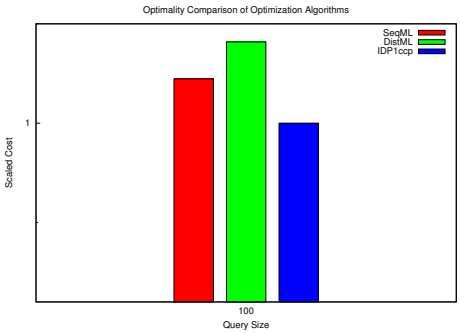
Graph 35 : Cycle, 40-Query



Graph 36 : Cycle, 60-Query

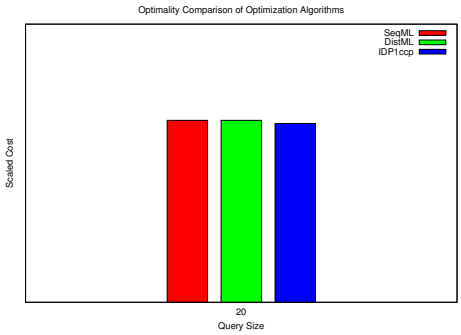


Graph 37 : Cycle, 80-Query

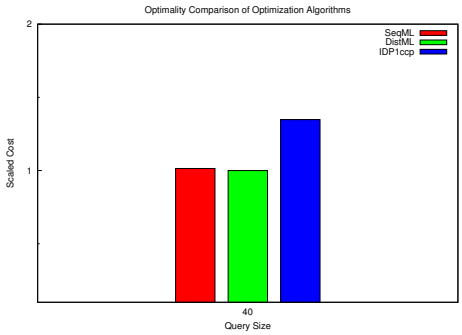


Graph 38 : Cycle, 100-Query

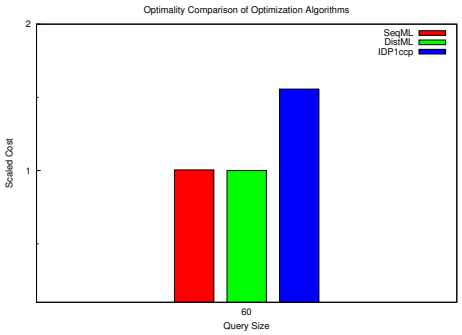
E.1.3 Clique Queries



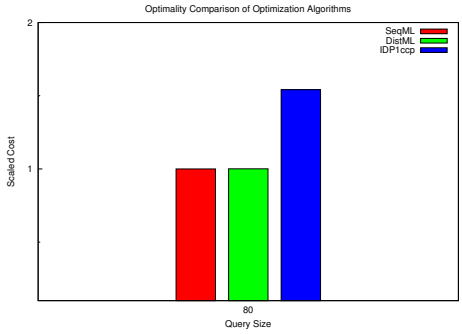
Graph 39 : Clique, 20-Query



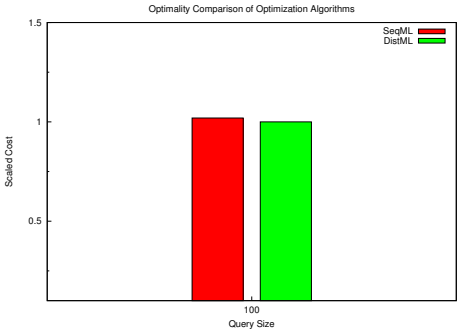
Graph 40 : Clique, 40-Query



Graph 41 : Clique, 60-Query

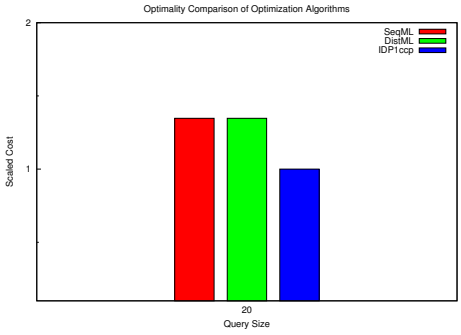


Graph 42 : Clique, 80-Query

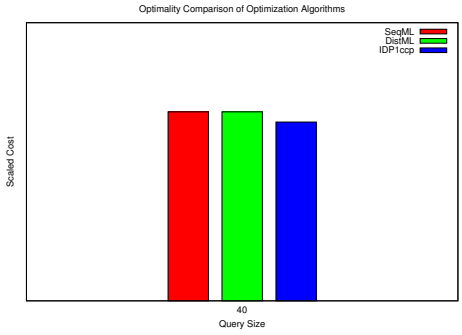


Graph 43 : Clique, 100-Query

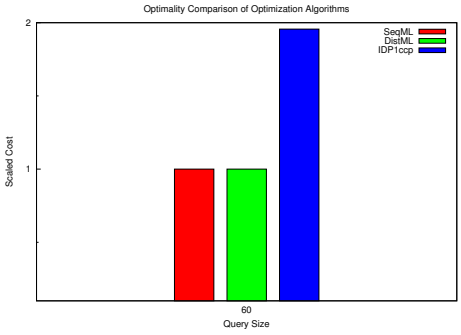
E.1.4 Star Queries



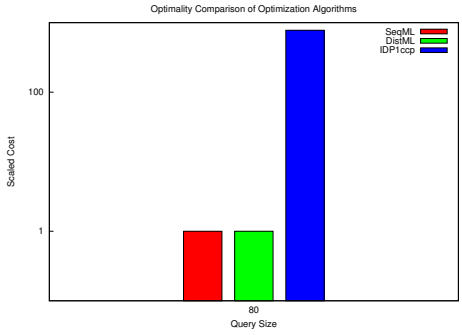
Graph 44 : Star, 20-Query



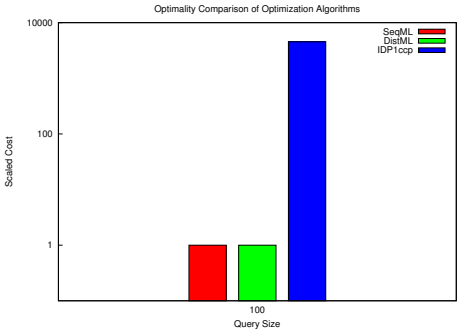
Graph 45 : Star, 40-Query



Graph 46 : Star, 60-Query



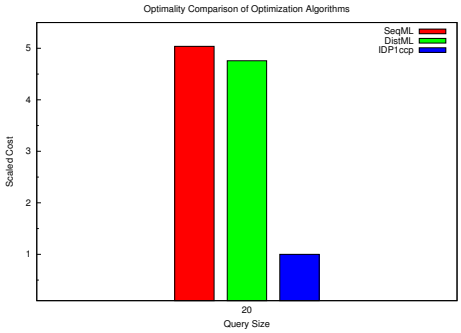
Graph 47 : Star, 80-Query



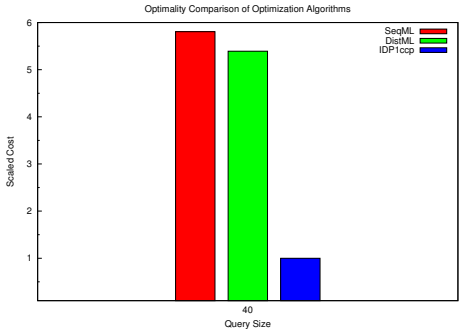
Graph 48 : Star, 100-Query

E.2 Distributed database with nine execution sites

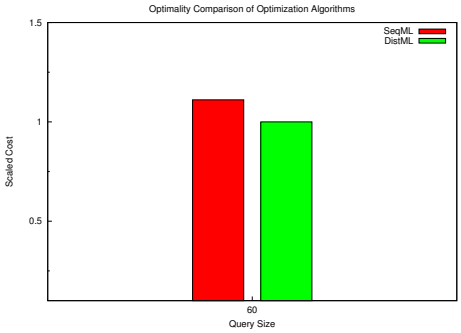
E.2.1 Chain Queries



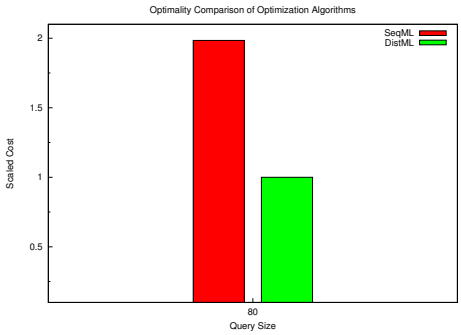
Graph 49 : Chain, 20-Query



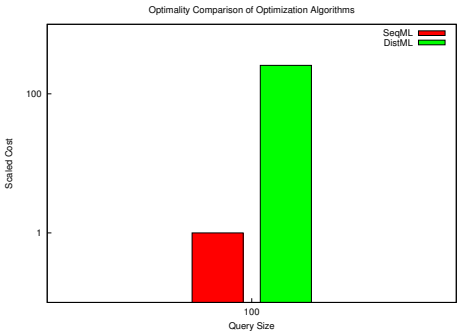
Graph 50 : Chain, 40-Query



Graph 51 : Chain, 60-Query

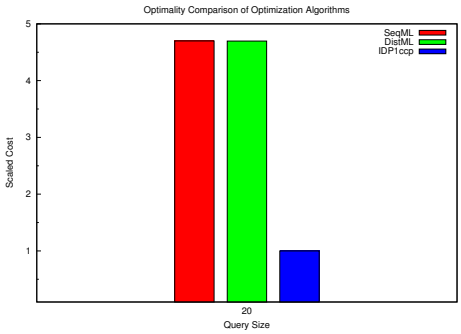


Graph 52 : Chain, 80-Query

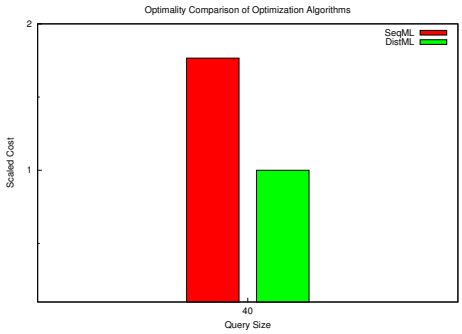


Graph 53 : Chain, 100-Query

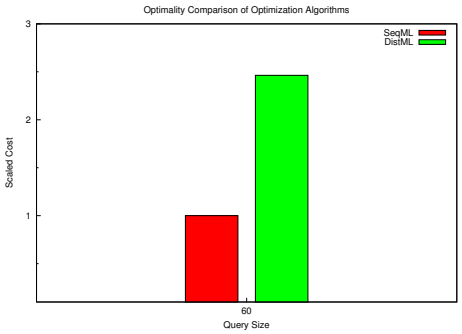
E.2.2 Cycle Queries



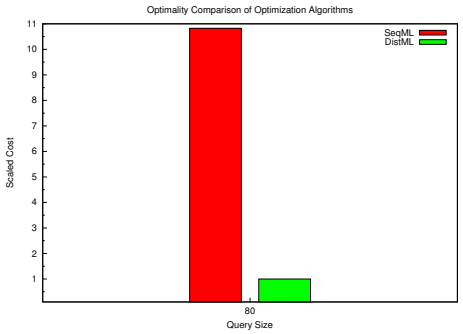
Graph 54 : Cycle, 20-Query



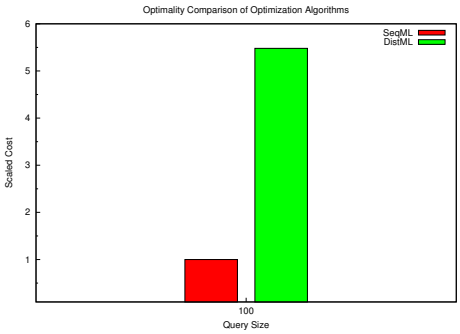
Graph 55 : Cycle, 40-Query



Graph 56 : Cycle, 60-Query

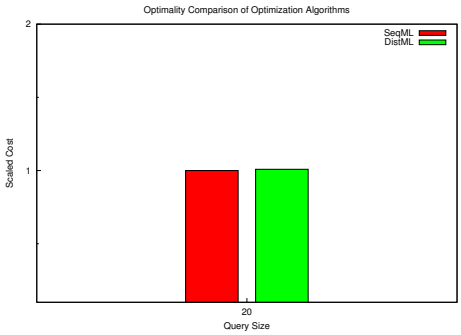


Graph 57 : Cycle, 80-Query

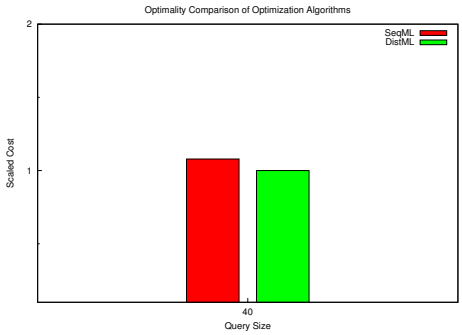


Graph 58 : Cycle, 100-Query

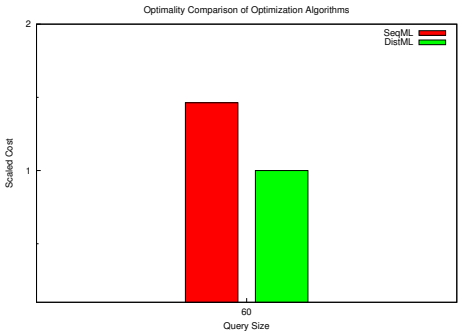
E.2.3 Clique Queries



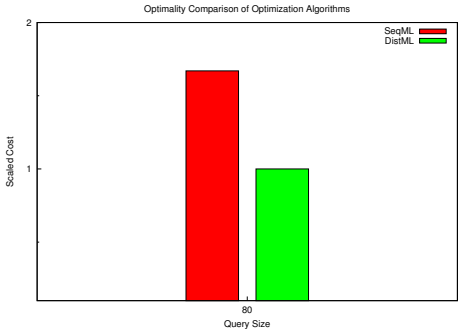
Graph 59 : Clique, 20-Query



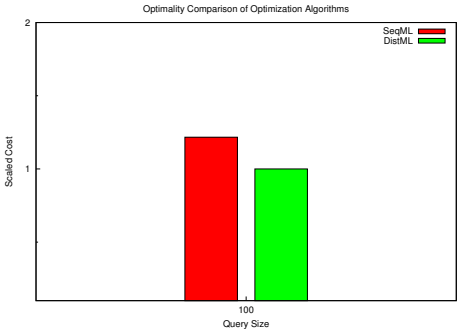
Graph 60 : Clique, 40-Query



Graph 61 : Clique, 60-Query

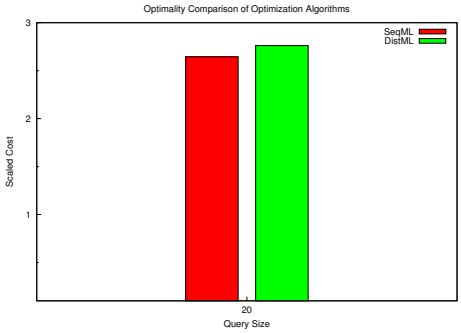


Graph 62 : Clique, 80-Query

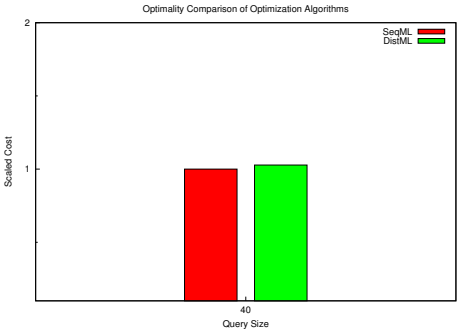


Graph 63 : Clique, 100-Query

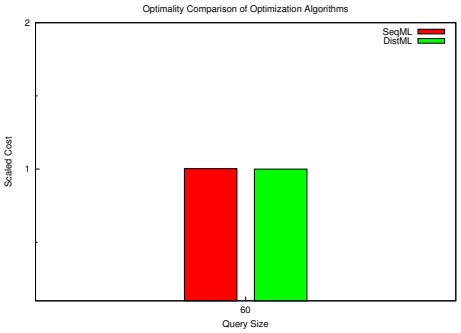
E.2.4 Star Queries



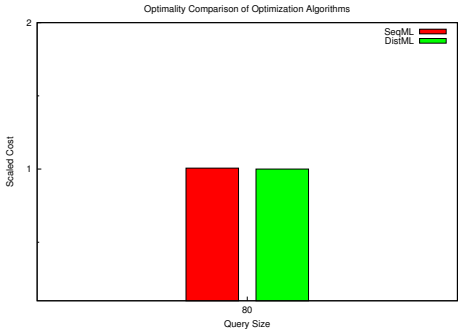
Graph 64 : Star, 20-Query



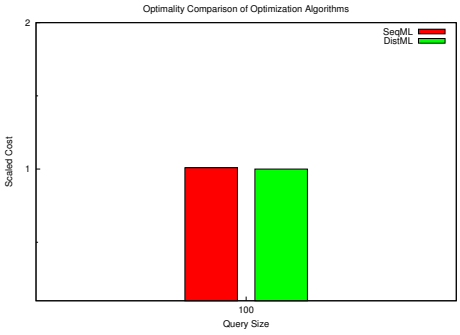
Graph 65 : Star, 40-Query



Graph 66 : Star, 60-Query



Graph 67 : Star, 80-Query



Graph 68 : Star, 100-Query

F Maximum K Values

Below, we present the maximum k values that can be used by each algorithm SeqML, DistML and IDP1ccp in order to comply with the 30 second time cap. '-' indicates that no k value could be found to meet the time cap.

Query Size	Query Type	1 Execution Site	3 Execution Sites	9 Execution Sites
20	Chain	20	20	4
20	Cycle	20	20	4
20	Clique	4	3	-
20	Star	6	4	2
40	Chain	40	15	2
40	Cycle	40	12	-
40	Clique	3	2	-
40	Star	4	3	2
60	Chain	60	9	-
60	Cycle	59	8	-
60	Clique	2	-	-
60	Star	3	2	2
80	Chain	79	7	-
80	Cycle	40	6	-
80	Clique	2	-	-
80	Star	2	2	2
100	Chain	40	5	-
100	Cycle	32	5	-
100	Clique	-	-	-
100	Star	2	2	-

Table 4: IDP1ccp Maximum k values

Query Size	Query Type	1 Execution Site	3 Execution Sites	9 Execution Sites
20	Chain	20	20	8
20	Cycle	20	19	8
20	Clique	11	8	4
20	Star	15	11	6
40	Chain	40	26	6
40	Cycle	40	19	6
40	Clique	10	7	4
40	Star	14	9	5
60	Chain	60	19	5
60	Cycle	59	19	5
60	Clique	10	6	3
60	Star	14	8	4
80	Chain	68	18	5
80	Cycle	68	16	5
80	Clique	10	6	3
80	Star	13	8	4
100	Chain	67	17	4
100	Cycle	61	16	4
100	Clique	9	6	3
100	Star	13	8	3

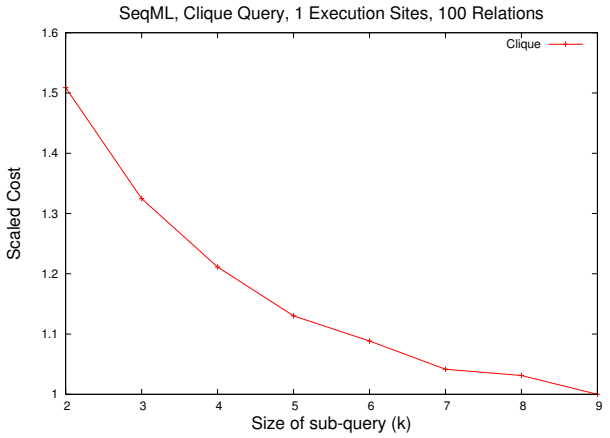
Table 5: SeqML maximum k values

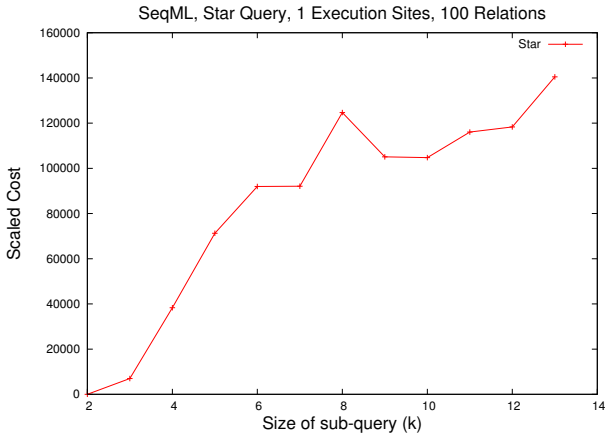
Query Size	Query Type	1 Execution Site	3 Execution Sites	9 Execution Sites
20	Chain	20	20	10
20	Cycle	20	19	10
20	Clique	11	8	5
20	Star	15	11	7
20	Mixed	11	8	5
40	Chain	40	26	10
40	Cycle	40	25	8
40	Clique	11	8	5
40	Star	15	10	6
40	Mixed	11	8	5
60	Chain	60	25	10
60	Cycle	59	24	8
60	Clique	11	8	5
60	Star	15	10	6
60	Mixed	11	8	5
80	Chain	68	25	10
80	Cycle	68	24	8
80	Clique	11	8	5
80	Star	15	10	6
80	Mixed	11	8	5
100	Chain	67	24	8
100	Cycle	67	23	8
100	Clique	11	7	4
100	Star	15	10	6
100	Mixed	11	7	4

Table 6: DistML maximum k values

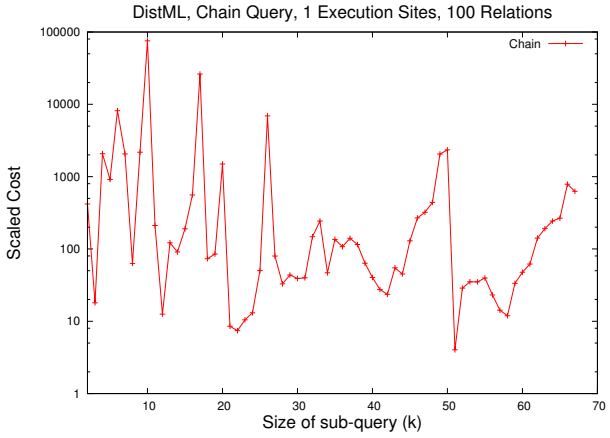
G Plan Quality with Varying K

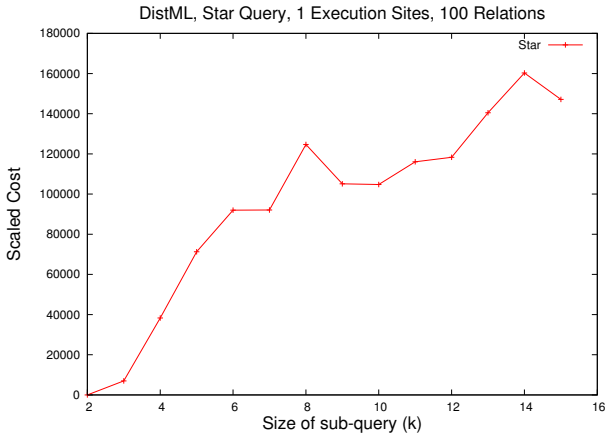
G.1 System with one Execution Site - SeqML - 100-Queries



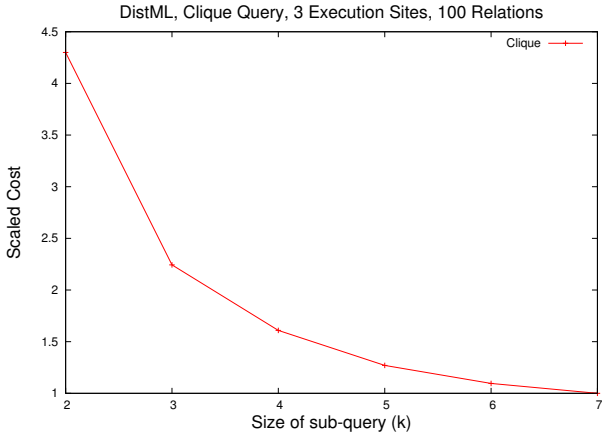


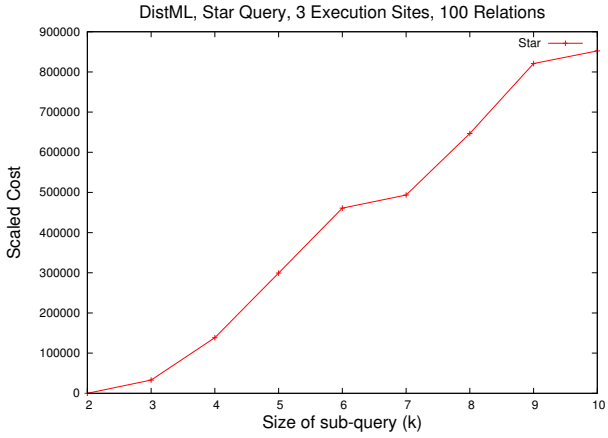
G.2 System with one Execution Site - DistML - 100-Queries



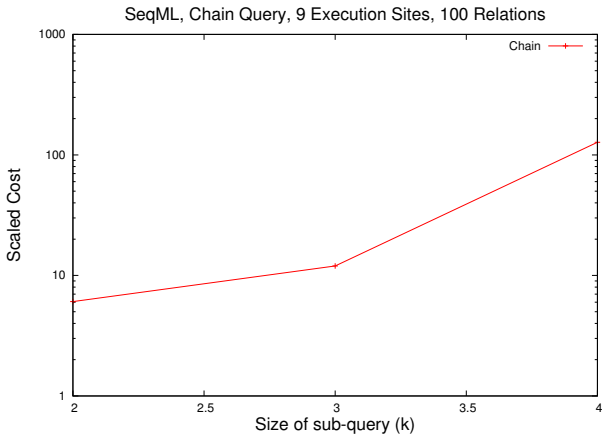


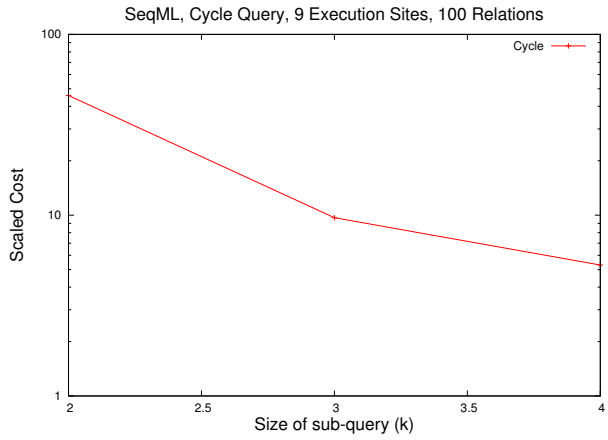
G.3 Distributed System with three Execution Sites - DistML - 100-Queries

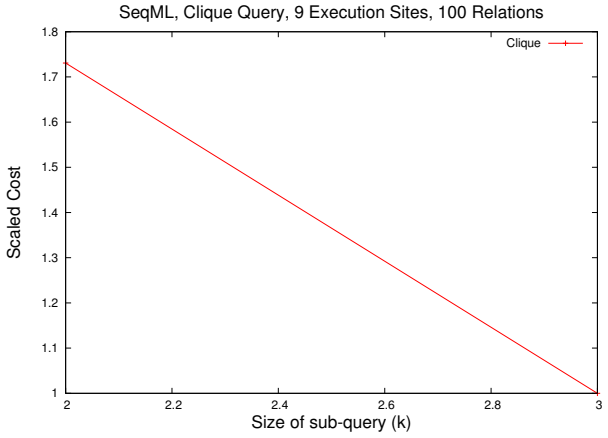


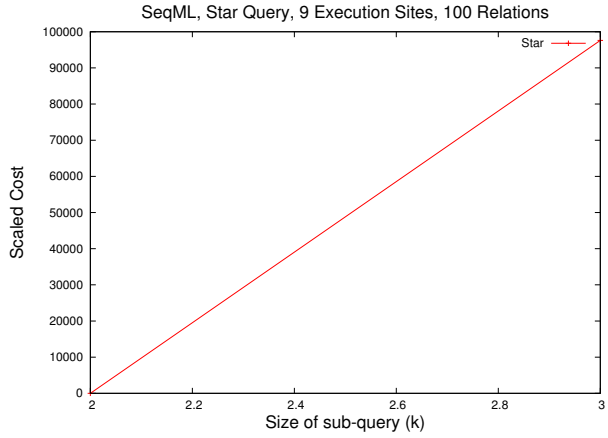


G.4 Distributed System with nine Execution Sites - SeqML - 100-Queries









G.5 Distributed System with nine Execution Sites - DistML - 100-Queries