Join Ordering with Learned Cardinalities

Enrique Moreno Caballero

4th Year Project Report
Artificial Intelligence and Computer Science
School of Informatics
University of Edinburgh

2022

Abstract

In this paper I explore recent machine-learning cardinality estimation techniques and analyse the performance of major join ordering algorithms with synthetic cardinalities of varying accuracy against the optimal dynamic programming algorithm. I implement the dynamic programming algorithm and three variations of greedy join ordering algorithms in Python from scratch. These algorithms are then used in my experiments, where I compare the join ordering plans produced by each of them with different cardinality estimates. The results show that the variation of cardinality accuracies do not produce a significant effect on the join orderings, although these orderings are far from optimal when compared with the dynamic programming join orderings.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

We are living in the data era. Since the beginning of the 21st century, the development of fast-growing digital technologies (lead by the Internet) has caused an exponential increase in the amount of data [1]. Digitalised information has become the driving force of the digital society [2]. The famous quote attributed to Sir Francis Bacon is more true than ever: "Knowledge is power".

In a technologically-driven society, businesses must learn to harness the data they collect in order to succeed. Data is always being collected, stored, and analysed. As a result, databases have become an essential tool for businesses and organisations around the world. For a database to be effective, it has to be managed wisely. Advantages of using databases in a business include reduced data redundancy, quick access to information or reduced data entry, storage, and retrieval costs.

The Structured Query Language (SQL) is the *lingua franca* for storing, manipulating and retrieving data in databases. SQL introduced a fundamental operation in relational database management systems (RDBMS) named **join**[1]. If we think of a relational database as a table made up of rows and columns, the join clause is used to combine rows from two or more tables, based on a related column between the tables. The goal of this project is to investigate the performance of different algorithms with database queries containing multiple joins and how the cardinalities (sizes) of the relations to be joined can influence the join plans produced by the algorithms.

1.2 Query Optimisation

Because SQL is a declarative language, the queries written by the users specify the desired result instead of how the query is to be executed. This is where query optimisers come into play. They are a key component of many RDMBS and are designed to figure out the most efficient way to execute a given query by considering the potential query

[1] `https://docs.microsoft.com/en-us/sql/relational-databases/performance/joins?view=sql-server-ver15`

plans. The order in which the joins are performed can have a significant effect on the amount of work required [3]. Since joins are so frequently used in such queries, the optimiser must be able to handle and order them intelligently. There are many different optimisation goals that are reasonable, such as minimising response time, minimising resource consumption, or minimising the time to obtaining the first output row [4].

When generating a join plan, query optimisers use intermediate results to decide which relations to join together such that the total cost is minimised [5]. Initially, the main goal of this project was to use a deep learning approach to learn the intermediate subquery sizes required by some join ordering algorithms. These subquery output sizes would then be fed to the join ordering algorithms. Finally, the performance of the join ordering algorithms with learned cardinalities would be evaluated against the join plans generated by a relational database management system. As described in Section 5.1, this goal was not achievable due to the lack of documentation on how to feed new (sub)queries to the cardinality estimator. Instead, the goal was changed to using synthetic selectivity ratios to evaluate the sensitivity of selectivity on the quality of generated join plans by some major join ordering algorithms.

1.3 Objectives

- Understand the challenges of the join ordering problem.
- Review of recent selectivity estimation techniques using machine learning.
- In-depth analysis of join ordering algorithms: Greedy Algorithms, Dynamic Programming.
- Understand the role of sub-query cardinality/selectivity ratios in these algorithms.
- Implementation of join ordering algorithms.
- Experimental evaluation of the performance of major join ordering algorithms with synthetic selectivities.
- Study the impact of feeding cardinalities with varying accuracy to the algorithms on the quality of join orderings.

1.4 Overview

As part of this project, I implemented four different join ordering algorithms in Python and studied the impact of different synthetic selectivity ratios on the quality of join ordering plans generated by these algorithms. For each set of selectivity ratios, I compared the join ordering returned by each algorithm using that set with the “optimal” join ordering computed by a dynamic programming algorithm using the same set. In this way, I was able to measure how “far away” the target algorithm behaves from the optimal dynamic programming method. The code for these implementations can be found in the project’s GitHub repository[2].

[2] `https://github.com/akathemix/Join-Ordering-with-Learned-Cardinalities`

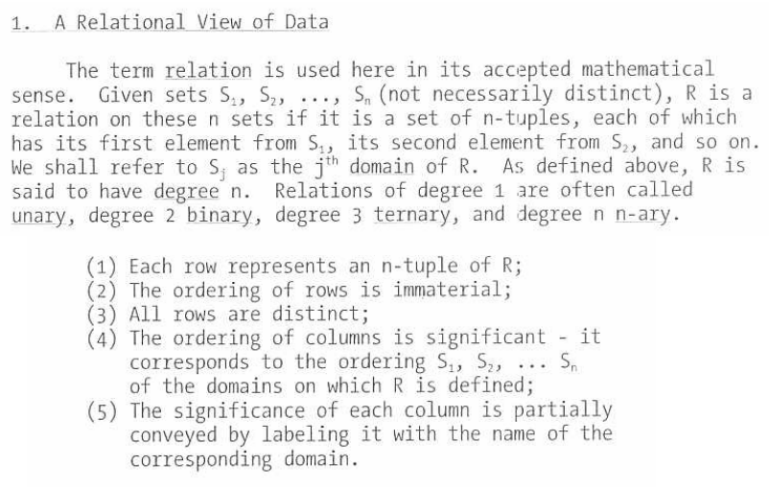
- Chapter 2 introduces some of the key concepts of the project in very simple terms, such as the JOIN clause, cardinalities and selectivities, or the join ordering problem. The chapter concludes with an explanation of the importance of having good cardinality estimates and promising machine learning techniques to do so.
- In Chapter 3, I discuss the greedy algorithms and the dynamic programming algorithm for join ordering.
- In Chapter 4, I present my implementations of the dynamic programming algorithm and three versions of a greedy-heuristic join ordering algorithm. I also explain the reasons behind my algorithm design decisions.
- Chapter 5 describes the experiments that were done on my implementations of the join ordering algorithms. I also mention a planned experiment that was not achievable.
- In Chapter 6 I evaluate the results of my experiments from Chapter 5.

Chapter 2

Background

2.1 Relational Model and SQL

The concept of the relational model for database management was first introduced by English computer scientist Edgar F. Codd in 1969 [6] [7]. The relational model is a data model based on predicate logic and set theory. The fundamental idea behind it is the relational view of data (see Figure 2.1). Relations can be thought of in logical form as datasets of tuples. More commonly, a relation is conceptualised as a table consisting of rows and columns. Each row represents a set of related data, whereas each column denotes a set of data values of a particular type (or **attribute**).



1. A Relational View of Data

The term relation is used here in its accepted mathematical sense. Given sets S_1, S_2, ..., S_n (not necessarily distinct), R is a relation on these n sets if it is a set of n-tuples, each of which has its first element from S_1, its second element from S_2, and so on. We shall refer to S_j as the j^{th} domain of R. As defined above, R is said to have degree n. Relations of degree 1 are often called unary, degree 2 binary, degree 3 ternary, and degree n n-ary.

(1) Each row represents an n-tuple of R;
(2) The ordering of rows is immaterial;
(3) All rows are distinct;
(4) The ordering of columns is significant - it corresponds to the ordering S_1, S_2, ... S_n of the domains on which R is defined;
(5) The significance of each column is partially conveyed by labeling it with the name of the corresponding domain.

Figure 2.1: Edgar F. Codd on the Relational View of Data [6]

The invention of the relational model was a turning point for the world of databases, as it became the foundation of the Structured Query Language (SQL) [8]. SQL is a domain-specific language used to manage and retrieve information from relational database management systems (RDBMS). According to the TIOBE[1] index (a measure of popularity of programming languages), SQL is the most popular database query

[1] https://www.tiobe.com/tiobe-index/

language and the tenth most popular programming language in the world (as of April 2021).

2.2 SQL Join Clause

To gain a better understanding of what the SQL (inner) join does, I will go through a brief example. The following example has relations **Customer** and **Account** which store a customer and bank account's details, respectively. Note that the **ID** column in **Customer** and the **CustomerID** column in **Account** represent the same thing, where each customer has their unique ID number.

ID	Name	City
1	Alice	Edinburgh
2	Bob	London
3	Claire	London

Table 2.1: **Customer**

AccountID	CustomerID	Balance
1234	3	25
5678	1	-100

Table 2.2: **Account**

Creating a new relation which contains every pair of rows from the two relations above does not provide a meaningful result. This is known as the cartesian product. The cardinality (number of rows) of the cartesian product of **Customer** and **Account** is the product of the two relations.

ID	Name	City	AccountNumber	CustomerID	Balance
1	Alice	Edinburgh	1234	3	25
1	Alice	Edinburgh	5678	1	-100
2	Bob	London	1234	3	25
2	Bob	London	5678	1	-100
3	Claire	London	1234	3	25
3	Claire	London	5678	1	-100

Table 2.3: Cartesian product of **Customer** and **Account**

To obtain a more meaningful relation between **Customer** and **Account**, it would be necessary to apply a filter (also known as predicate) to the cross product of these two relations. Now let's imagine the goal is to find out the names and balances of those customers whose balance is below zero. The SQL query would look something like Figure 2.2. Note that using **JOIN ON** and **WHERE** together has been found to perform 6% faster than using a unique **WHERE** clause [9]. The output of the query can be seen in Table 2.4.

```
SELECT Name, Balance
FROM   Customer, Account
WHERE  ID = CustID AND Balance < 0
```

or alternatively:

```
SELECT Name, Balance
FROM   Customer JOIN Account ON ID=CustID
WHERE  Balance < 0
```

Figure 2.2: Join **Account** and **Customer** on ID=CustID

Name	Balance
Alice	-100

Table 2.4: **Query Output**

2.3 Join Ordering Problem

Inner joins are always commutative [10], regardless of the number of inner joins performed. This means that any order of executing the inner joins will obtain the same result. This happens because the attributes (columns) in the relational model do not have any specific order. Therefore, the cross products can occur in any order. The same reasoning follows for associativity [10]. If a relation is to be filtered by a predicate, it does not matter whether that relation is used in cross products first. As long as the filtering takes place at some point, the result will be the same.

The commutative and associate properties of the inner join are one of the key elements of the join ordering problem. Given a join query Q consisting of relations R_1,..., R_n, the join ordering problem is to find the best join plan for Q that joins R_1,..., R_n together such that the total join cost is minimised.

2.3.1 Cardinalities and Selectivities

The query optimisation procedure must achieve a less-costly solution than trying all possible orderings. To do this, it is possible to use some of the information available about the relations to order. To gain an understanding of how much a predicate filters a query result, a numeric value called the predicate's selectivity can be calculated. The selectivity of a predicate p on relations A and B can be calculated by dividing the size (cardinality) of the filtered cross-product of A and B by the size of the unfiltered cross-product of A and B (as shown in Figure 2.3).

However, the formula in Figure 2.3 assumes that the size of the filtered result is known already, when in fact the idea is to use the predicate's selectivity to estimate the cardinality of the filtered result (as shown in Figure 2.4). This is the classic problem of the chicken and the egg. It is not possible to know the exact value of one before the value of the other is known, and viceversa.

By looking at the formula in Figure 2.4), it is straightforward that predicates with a

$$sel(p) = \frac{|A \bowtie_p B|}{|A \times B|} = \frac{|A \bowtie_p B|}{|A||B|}$$

Figure 2.3: Formula to calculate selectivity of a predicate

$$|A \bowtie_p B| = sel(p)|A||B|$$

Figure 2.4: Using the predicate selectivity to estimate cardinality of the join result

lower selectivity value will filter database rows more aggressively, hence decreasing the cardinality of the query output. A common principle is to use these more selective joins first to reduce the output cardinality for the subsequent joins. This will avoid computing big subresults that are not needed afterwards, although this will not always result in the optimal join ordering plan.

2.4 Cardinality Estimation in RDBMS

In the 2015 paper "*How Good Are Query Optimizers, Really?*" [3], the authors investigate the quality of industrial-strength cardinality estimators and find that all of these estimators routinely produce large errors. They show that "*while estimates are essential for finding a good join order, query performance is unsatisfactory if the query engine relies too heavily on these estimates*". The results of their experiments show that relational database systems produce large estimation errors that quickly grow as the number of joins increases, and that these errors are usually the reason for bad plans. They also show that cardinality estimation is the most important factor in producing good query plans and propose that database systems should incorporate more advanced estimation algorithms.

2.5 Machine Learning for Selectivity Estimation

In 2018, three years after the publication of "*How Good Are Query Optimizers, Really?*" [3], some of the researchers of the paper came up with an interesting paper on using supervised learning to solve cardinality estimation in isolation. They consider cardinality estimation to be "*the Achilles heel of database query optimization*" [11].

In this paper, the authors argue that machine learning is a highly promising technique for solving the cardinality estimation problem, with the input being query features and the output being the estimated cardinality. They are not aiming for perfection, simply for the ML-based estimator to perform better than their baseline estimator. The researchers in the paper used a specialised deep learning model called multi-set convolutional network (MSCN) and evaluated their approach with the real-world IMDB

dataset.

A brief overview of the ML approach is that once a supervised learning algorithm has been trained with query/output cardinality pairs, the model can be used to estimate unseen queries. One key challenge of this learning task is how to train the model before having concrete information about the query workload. Their approach is to obtain an initial training corpus by generating random queries based on the schema information and drawing literals from actual values in the database. A training sample consists of table identifiers, predicates, and the true cardinality of the sample query result. The training data knows about materialised base table samples. For each table in a query, the corresponding predicates on a materialised sample are evaluated. Then each query is annotated with the number of qualifying samples for this table. These same steps are performed for an unseen test query at estimation time, thus allowing the ML model to utilise this knowledge.

An advantage of this ML approach is that it is very cheap in query time. The disadvantages are that it cannot handle all relevant filter types and the estimates become stale over time [12].

Chapter 3

Analysis of Join Ordering Algorithms

3.1 Greedy Algorithms

3.1.1 Greedy Join Ordering (Version 1)

For each relation R to be joined, the weight function for R is its cardinality $|R|$. At each step, this algorithm chooses the less-costly relation that has not been chosen already and adds it to the join ordering sequence. This step is repeated until there are no relations to choose from. This algorithm always makes locally optimal choices and may get stuck in local optima. One disadvantage of this algorithm is the use of fixed weight functions that do not change with respect to the relations chosen already. This is something which can be changed, but the first relation to be chosen will still have a huge effect on the result of the algorithm.

```
GreedyJoinOrdering-1({R_1,...,R_n}, (*weight)(Relation))
Input:  a set of relations to be joined and a weight function
Output:  a join order
S = ϵ; // initialize S to the empty sequence
R = {R_1,...,R_n}; // let R be the set of all relations
while(!empty(R)) {
  Let k be such that:  weight(R_k) = min_{R_i∈R}(weight(R_i));
  R\ = R_k; // eliminate R_k from R
  S○ = R_k; // append R_k to S
}
return S
```

Figure 3.1: Pseudocode for Greedy Join Ordering (Version 1) [13]

3.1.2 Greedy Join Ordering (Version 2)

As in Version 1, the second version of the greedy algorithm looks at the relations that are not in the order plan yet and decides which is the best relation to join next. As opposed to Version 1, this decision is not based solely on a relation's cardinalty. Instead, Version 2 chooses to join on the relation that produces the smallest intermediate

result when joined with the current order plan. The weight function here differs from that in Version 1 since each relation in Version 2 also takes into account the join order sequence computed so far. The problem of the impact on the result of the first relation to be joined is still present here, but Version 3 presents a solution to overcome it.

```
GreedyJoinOrdering-2({R_1, ..., R_n},
            (*weight)(Sequence of Relations, Relation))
Input:  a set of relations to be joined and a weight function
Output:  a join order
S = ϵ; // initialize S to the empty sequence
R = {R_1, ..., R_n}; // let R be the set of all relations
while(!empty(R)) {
  Let k be such that:  weight(S, R_k) = min_{R_i∈R}(weight(S, R_i));
  R\ = R_k; // eliminate R_k from R
  S○ = R_k; // append R_k to S
}
return S
```

Figure 3.2: Pseudocode for Greedy Join Ordering (Version 2) [13]

3.1.3 Greedy Join Ordering (Version 3)

If there are *n* relations to be joined, the third version of the greedy algorithm runs Version 2 *n* times. At each iteration, Version 3 starts the join order sequence with a different relation. At the end, the algorithm will have *n* join orderings and will output the ordering which minimises the selectivities. This heuristic is also known as *MinSel* [14].

```
GreedyJoinOrdering-3({R_1, ..., R_n}, (*weight)(Sequence of Relations, Relation))
Input:  a set of relations to be joined and a weight function
Output:  a join order
Solutions = ∅;
for (i = 1; i ≤ n; ++i) {
  S = R_i; // initialize S to a singleton sequence
  R = {R_1, ..., R_n} \ {R_i}; // let R be the set of all relations
  while(!empty(R)) {
    Let k be such that:  weight(S, R_k) = min_{R_i∈R}(weight(S, R_i));
    R\ = R_k; // eliminate R_k from R
    S○ = R_k; // append R_k to S
  }
  Solutions += S;
}
return cheapest in Solutions
```

Figure 3.3: Pseudocode for Greedy Join Ordering (Version 3) [13]

3.2 Dynamic Programming

The basic premises of the dynamic programming approach for the join ordering problem are the optimality principle and avoiding duplicate work. The optimality principle

is that whatever the initial state is, remaining decisions must be optimal with respect to the state following from the first decision. [15]

To find a join ordering plan, dynamic programming is often implemented with a cost table in which only good sub-solutions are remembered. The table stores the estimated sizes of the join results, the estimated lowest costs for performing the join and the expression which produced the lowest cost. I highly recommend going through this intuitive, step-by-step demonstration[1] by the Department of Math and Computer Science from Emory University to grasp a better understanding of the dynamic programming algorithm for join ordering.

[1] `http://www.mathcs.emory.edu/~cheung/Courses/554/Syllabus/5-query-opt/dyn-prog-join3.html`

Chapter 4

Implementation of Join Ordering Algorithms

4.1 Design Decisions

The implementations of the greedy (versions 1,2,3) and dynamic programming algorithms follow are avalaible in the project's associated GitHub repository[1]. These implementations are based on their corresponding pseudocode from Chapter 3.

4.1.1 Join Order Benchmark (JOB)

The algorithms use the Join Ordering Benchmark (JOB) queries from "*How Good Are Query Optimizers, Really?*" [3] that are available in GitHub[2]. This benchmark is based on the IMDB[3] dataset. The benchmark has an average of 8 joins per query, where each query consists of a select-project-join block. The large amount of correlations in this dataset can be a problem for most cardinality estimators.

4.1.2 Workload: Scale

For evaluation, a synthetic workload named "*scale*" is used. It contains 500 queries and is designed to show how the cardinality estimators behaves with up to 4 joins. I chose *Scale* over the *Synthetic* and *JOB-light* workloads[4] as the distribution of joins in *Scale* (see Figure 4.1) was uniformly spread and because it contained the largest amount of queries with more than 2 joins out of the workloads. These are the queries of interest where we can really observe the different performance of the join ordering algorithms.

[1] https://github.com/akathemix/Join-Ordering-with-Learned-Cardinalities
[2] https://github.com/gregrahn/join-order-benchmark
[3] https://www.imdb.com/
[4] https://github.com/andreaskipf/learnedcardinalities/tree/master/workloads

number of joins	0	1	2	3	4	overall
synthetic	1636	1407	1957	0	0	5000
scale	100	100	100	100	100	500
JOB-light	0	3	32	23	12	70

Figure 4.1: Distribution of joins across workloads [11]

4.1.3 Weight Function: Cardinalities of Relations

The algorithm implementations rely on the ***get_cardinalities*** method[5]. This method takes in queries and assigns a weight function to each relation in each query. The output of the method is a dictionary of dictionaries. The first keys are the indices *i* of the "*scale*" queries, ranging from 1 to 500. The second keys (key for each query *i*) are the names of the relations present in each query *i* and their corresponding values are their cardinalities.

I have chosen the weight function of each relation to be its own cardinality. The reason behind this is that the cardinalities of the columns of each relation are known[6].

In the cases where there is a constant selection applied on a column, a pre-processing operation takes place in which these constant selections are pushed down to the corresponding relations [16]. As the selectivity of each predicate is unknown, the ***get_cardinalities*** function will assign a random cardinality to each filtered relation with an upper bound on the original cardinality. If there is more than one predicate for columns of the same relation, then the most selective predicate is used. This decision of prioritising more selective predicates can have a significant effect in speeding up the join ordering process [12]. After that, the filtered relations become new relations that replace their previously unfiltered version, such that the query is equivalent and no longer has any constant selections. The cardinality of the filtered relation will be lower than that of the unfiltered relation, unless all the rows in the relation meet its predicates.

4.2 Algorithms

All the algorithm implementations described below follow the same structure:

- Take the 500 queries from the *scale* workload.
- Compute the base cardinalities for each relation in the query (using ***get_cardinalities***).
- If the number of joins in a query is less than 2, no calculation of join ordering is needed.

[5] https://github.com/akathemix/Join-Ordering-with-Learned-Cardinalities/blob/master/greedy1.py

[6] https://github.com/andreaskipf/learnedcardinalities/blob/master/data/column_min_max_vals.csv

- Otherwise, calculate join order plan for each query and store as a list (0-indexed), where the element at index 0 denotes the first relation of the join order plan, the index 1 denotes the second relation, and so on.
- Output dictionary of join order plans where each plan (list) is indexed by the query ID number (1-500).

Returning the join order plans in the same (dictionary) format across all algorithms was a crucial part to facilitate the comparison and evaluation of the join orderings (see Chapter 6).

4.2.1 Greedy Join Ordering (Version 1)

The first version of the greedy join ordering algorithm is the most simple among the implementations. The output is a dictionary that stores the join ordering plans for queries 1-500.

For each query, first, the relations included in the query are kept in a dictionary with relation (key) and cardinality (value) pairs. Then a new list is initialised. This list will keep track of the order plan. While there are still relations that have not been added to the join plan, the algorithm selects the relation with the minimum cardinality, adds it to the order plan, and removes it from the dictionary of remaining relations. Once there are no more relations to be added to the order plan, the order plan is added to the dictionary of join ordering plans and is indexed by the query's number identifier.

4.2.2 Greedy Join Ordering (Version 2)

In the second version[7] of the greedy algorithm, the weight function for each relation now takes into account the order plan computed so far. Note that for queries joining 2 relations, there is no need to do any calculations with respect to the relations' cardinalities since they can only be joined together. The first relation added to the join order plan is the one with the minimum cardinality. This cardinality is initially stored as the current cost of the join order plan.

Once the order plan contains a starting relation, the algorithm estimates the cardinality of joining each relation individually to the plan so far. To do this, I assign a random join cardinality with an upper bound on the cross product of the current order plan and the relation. Then the smallest of the possible intermediate results is selected and the relation-to-be-joined that caused this result is added to the join plan (and removed from the relations-to-be-joined). The cost of the join order plan is updated with the cost of this newly-computed intermediate result. The process is repeated until there are no remaining relations waiting to be joined.

[7] `https://github.com/akathemix/Join-Ordering-with-Learned-Cardinalities/blob/master/greedy2.py`

4.2.3 Greedy Join Ordering (Version 3)

The code for Version 3[8] of the greedy join ordering algorithm is very similar to that of Version 2. The main difference is that Version 3 computes a join order plan using every relation as the starting relation. This means that for a query with *n* relations, Version 3 will have *n* alternative join ordering plans to select from. It is therefore necessary to keep track of the costs of the multiple plans until the very end of the algorithm, where the algorithm will return the least-costly plan among the potential plans for each query.

4.2.4 Dynamic Programming

The dynamic programming implementation[9] was probably the most challenging, due to the complexity of finding an appropriate way to index the query sub-plans effectively. I will explain how I achieved this.

With the DP algorithm, for a query with *n* relations, there are *n*-choose-*k* possible join order plans of length *k*. Let's go through an example with *n=4* relations to be joined. These are *R*, *S*, *T* and *U*. Initially there are 4-choose-2 ways of getting a join plan of length 2: (*R*-*S*, *R*-*T*, *R*-*U*, *S*-*T*, *S*-*U*, *T*-*U*). Since inner joins are commutative, the join (*R*-*S*) is equivalent to (*S*-*R*). To avoid duplicate and equivalent orderings, the orderings are stored in a dictionary where the key for each ordering is made up of the ordering's relations sorted alphabetically and separated by a comma. The comma helps to differentiate relation names with more than one character.

Key	Value
R,S	[R,S]
R,T	[R,T]
R,U	[R,U]
S,T	[S,T]
S,U	[S,U]
T,U	[T,U]

Table 4.1: DP - Joining 2 relations

From the table above, the join of *R* and *S* (*R*,*S*) can be joined with either *T* (*R*,*S*,*T*) or *U* (*R*,*S*,*U*). To join *i* relations, you first look-up the combinations of length *i*-1 of those *i* relations and find out which join combination gives the least-costly result.

Let's assume that for relations (*R*,*S*,*T*), the least-costly sub-join is (*S*,*T*). Then the key-value pair for (*R*,*S*,*T*) when joining 3 relations would be updated to [S,T,R]. Note that the key is maintained in alphabetical order. Once the orderings (values) for all joins (keys) of length *i* are calculated, the joins of length *i*-1 are no longer needed. The process is repeated until the final join ordering of length *n* is calculated.

[8] https://github.com/akathemix/Join-Ordering-with-Learned-Cardinalities/blob/master/greedy3.py

[9] https://github.com/akathemix/Join-Ordering-with-Learned-Cardinalities/blob/master/dynamic_programming.py

Key	**Look-up**
R,S,T	[(R,S), (R,T), (S,T)]
R,S,U	[(R,S), (R,U), (S,U)]
R,T,U	[(R,T), (R,U), (T,U)]
S,T, U	[(S,T), (S,U), (T,U)]

Table 4.2: DP - Example of looking for least costly sub-plan following 4.1

Chapter 5

Experimentation

5.1 Learned Selectivity Ratios

One of the initial goals of the project was to compare the performance of major join ordering algorithms with learned selectivity ratios against the performance of RDBMS generated join plans. To generate the ML-based sub-query selectivity estimates for the implementations of the join ordering algorithms, I was going to use the multi-set convolutional network (MSCN) described in the 2018 paper "*Learned Cardinalities: Estimating Correlated Joins with Deep Learning*" [11]. The authors provide a PyTorch implementation of this MSCN that is publicly available in Github[1]. The cardinality estimator requires two things:

- A CSV file with a special formatting where each row represents an SQL query.
- Bitmaps representing the positions of the qualifying samples for each table in the query.

The above paper's original CSV files and bitmaps can also be found in the paper's associated repository[2]. Unfortunately, I had to deviate from this goal due to the lack of documentation on generating bitmaps for new SQL queries. I spent a long time writing a program[3] that took an SQL query and then generated its possible sub-queries. This was specially challenging time because it involved a reverse-engineering task. First, I had to learn how to read the CSV special format before being able to write my own SQL queries in the same format to a CSV file. After succeeding to generate my own CSV file with the relevant subqueries for each input query, I did not find any guidance on how to create compatible bitmaps for my subqueries.

[1] `https://github.com/andreaskipf/learnedcardinalities`

[2] `https://github.com/andreaskipf/learnedcardinalities/tree/master/workloads`

[3] `https://github.com/akathemix/Join-Ordering-with-Learned-Cardinalities/blob/master/possible_orderings.py`

5.2 Selectivity Ratios with Varying Accuracy

The main experimental[4] component of the project was to use synthetic cardinalities to evaluate the sensitivity of cardinalities on the quality of generated join plans by the implementations of the join orderings algorithms. For each set of synthetic cardinalities, I compare the join ordering returned by my implementations with these cardinalities against the "optimal" join ordering computed by a dynamic programming using the same cardinalities. By doing this, I measured how "far away" the target method behaves from the optimal method which is dynamic programming.

For each algorithm, these are the steps I followed:

- Vary the percentage of subqueries whose cardinality are modify.
  The percentages were: 10%, 20%, 25%, 33%, 50%, 75%.
- And also vary the percentage by which each subquery's cardinality is modified.
  The percentages were: 25%, 50%, 75%, 125%, 150%, 200%, 1000%.

[4] `https://github.com/akathemix/Join-Ordering-with-Learned-Cardinalities/blob/master/compare_results.py`

Chapter 6

Evaluation

In order to achieve a fair experiment, each experiment was performed with 10 different random seeds (fed into the algorithms). The random seeds affected the random cardinalities produced by the ***get_cardinalities***[1] method. By doing this, I got broader results of how the algorithms generalise to different cardinalities.

6.1 Comparing Performance of the Greedy Algorithms

Table 6.1 shows the results of comparing the greedy algorithms' join orderings (without modifying the cardinalities yet). From these results, it is clear that the choice of the first relation to join has a significant effect on the join order performance. Greedy1 and Greedy2 always choose the relation with the smallest cardinality as the starting relation, hence the similarity (almost 80% of their join order plans match). Greedy3 does not follow the same criteria to choose its starting relation and the percentage of matches drops to almost 50%.

ALGORITHM	AVERAGE MATCH
Greedy1 vs Greedy2	79%
Greedy1 vs Greedy3	55%
Greedy2 vs Greedy3	54%

Table 6.1: Average match between each greedy algorithm

6.2 Baseline: Standard Algorithms vs DP

The baseline experiment involved running the algorithms with the unmodified cardinality ratios and compare their performance with the dynamic programming algorithm. The results in Table 6.2 show the average match between join orderings of the greedy algorithms and the join ordering produced by the DP algorithm.

[1] `https://github.com/akathemix/Join-Ordering-with-Learned-Cardinalities/blob/master/greedy1.py`

ALGORITHM	AVERAGE MATCH WITH DP
Greedy 1	60%
Greedy 2	58%
Greedy 3	54%

Table 6.2: Average match between join orderings of the greedy algorithms and the join ordering produced by the DP algorithm.

% of queries with modified cardinalities	Modification of cardinalities	% of matches
50%	x0.25	56.8%
50%	x0.5	57.2%
50%	x1	57%
50%	x1.5	57.4%
50%	x2	56.8%
50%	x10	57.6%

Table 6.3: Average match between join orderings of the Greedy2 algorithm and DP algorithms – Using varying cardinality accuracies and 50% of queries being modified.

6.3 Varying Cardinalities

Table 6.3 shows an example of the results obtained by varying cardinality accuracies with 50% of the queries having their relations' cardinalities modified. When the cardinalities are not altered, the match is 57% (x1). We observe that the variation of accuracies does not produce any significant change in the percentage of join orderings that match between Greedy2 and DP. Table 6.4 shows the average results of feeding cardinalities with all the different varying accuracies to the greedy algorithms. The results are almost identical to those in 6.2, showing that varying the accuracy of the cardinalities used by the greedy algorithms does not change the join ordering plans it produces significantly.

ALGORITHM	AVERAGE MATCH
Greedy2	57%
Greedy3	55%

Table 6.4: Average match between join orderings of the greedy algorithms with varying cardinality accuracies and the join ordering produced by the DP algorithm.

Chapter 7

Conclusions

I have presented the join ordering problem, one of the most important challenges of query optimisation. I have explored different join ordering algorithms and analysed how they tackle the join ordering problem. Implementing the greedy algorithms and the dynamic programming algorithm allowed me to perform experiments on the role of sub-query cardinality ratios in these algorithms. The results of these experiments show that there is still a lot of room for improvement for the performance of join ordering algorithms. The future is bright, as the emergence of promising machine learning techniques for cardinality estimation will lead to better query plans by the RDBMS query optimisers.

Bibliography

[1] Li Cai and Yangyong Zhu. The challenges of data quality and data quality assessment in the big data era. *Data Sci. J.*, 14:2, 2015.

[2] Sora Park. *Information is Power*, pages 161–183. 11 2017.

[3] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, 2015.

[4] Database performance and query optimization. `https://www.ibm.com/docs/en/i/7.4?topic=database-performance-query-optimization`.

[5] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '98, page 34–43, New York, NY, USA, 1998. Association for Computing Machinery.

[6] E. F. Codd. Derivability, redundancy and consistency of relations stored in large data banks. *Research Report / RJ / IBM / San Jose, California*, RJ599, 1969.

[7] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[8] Alan Bivens. Impact of the SQL Relational Model 50 years later. `https://www.ibm.com/blogs/research/2020/06/sql-relational-model-50-years-later/`.

[9] Fawwaz Alnawajha. The performance of inner join types in sql. 01 2016.

[10] Anil Shanbhag and S. Sudarshan. Optimizing join enumeration in transformation-based query optimizers. *Proc. VLDB Endow.*, 7(12):1243–1254, 2014.

[11] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677*, 2018.

[12] Axel Hertzschuch, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. Simplicity done right for join ordering. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021.

[13] Guido Moerkotte. *Building Query Compilers (Under Construction) [expected time to completion: 5 years]*. 10 2009.

[14] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Optimizing join orders. 12 1999.

[15] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, 1989.

[16] Laurel J. Orr, Srikanth Kandula, and Surajit Chaudhuri. Pushing data-induced predicates through joins in big-data clusters. *Proc. VLDB Endow.*, 13(3):252–265, 2019.