Master's Thesis

An Efficient Query Planner for the QLever SPARQL Engine

Mahmoud Khalaf

(Matrikelnummer: 4974461)

Examiner: Prof. Dr. Hannah Bast
Prof. Dr. Christian Schindelhauer

Adviser: Johannes Kalmbach

Albert-Ludwigs-Universität Freiburg

Technische Fakultät

Institut für Informatik

Professur für Algorithmen und Datenstrukturen

January 18, 2025

Writing Period

18. 07. 2024 – 18. 01. 2025

Examiner

Prof. Dr. Hannah Bast

Second Examiner

Prof. Dr. Christian Schindelhauer

Adviser

Johannes Kalmbach

Declaration

I hereby declare that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I hereby also declare that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

_____ _____

Place, Date Signature

Abstract

Handling increasingly complex queries is not only expectation, but the main prerequisite for any robust modern relational database management system. As extracting any meaningful insight requires interacting with dozens and dozens of relations in a single query.

Having a domain expert synthesize a hand-crafted query in SPARQL is considered the exception and not the rule, as the vast majority of the queries in the modern day and age are machine-generated using high-level frontends, dashboards and business intelligence tools. Synthesizing a good concrete execution plan is the relational database management system's Query Planner require joining multiple relations. the Join order of the relations involved drastically affect the quality of the final plan.

Optimal Join Order of a set of relations can be formulated as an NP-Hard Optimal Scheduling problem [1]. the same approaches, techniques and methodologies will be repurposed for our intended use case. Exploring the search space of all possible plans while constrained by a reasonable time frame and a computational budget heavily relay on using estimation techniques to guess plan cost and using clever heuristics to prune the search space. This thesis will mainly focus on Join Ordering techniques capable of synthesizing efficient query plans where relatively large number of relations are involved in polynomial time complexity.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

'Before we reach our goal,' the hoopoe said,
'The journey's seven valleys lie ahead;

The Conference of the Birds

1.1 Motivation

The amount of data generated on the world wide web has been on the raise and showing no signs of slowing down anytime soon, which begs for quick information retrieval systems that can keep-up with the ever-increasing appetite. Knowledge Graphs (section 1.2) has been the go-to approach for representing aforementioned information. this thesis will explore methods, tools and techniques to quickly retrieve information from a properly engineered SPARQL (section 1.4) database management system.

1.2 Knowledge Graph

Knowledge graphs [3] are directed graphs with labeled edges that represent structured knowledge about the world. Each vertex stands for an entity. Each directed edge stands for a relation between two entities and the label says what the relation is.

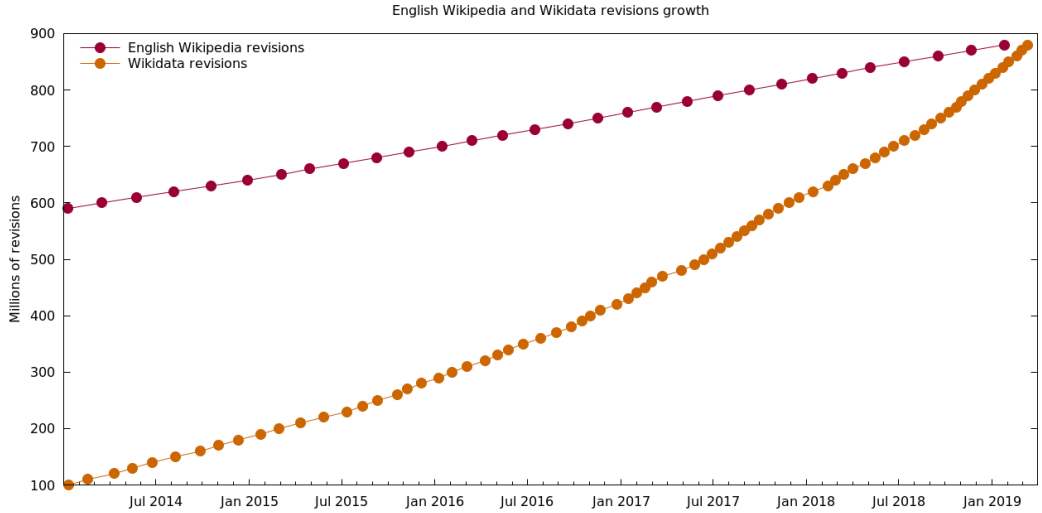


Figure 1: Wikidata revisions, 2014-2019 [2]

The World Wide Web has brought us vast amounts of data in electronic form and the possibility of crowdsourcing. As a consequence, the field has been reborn and many large knowledge graphs have been developed over the past fifteen years. Some contain general-purpose knowledge, like Freebase, Yago, DBpedia, or Wikidata. Others contain domain-specific knowledge, like UniProt (proteins), PubChem (chemistry), OpenStreetMap (geodata), or DBLP (bibliographic data).

1.3 Resource Description Framework (RDF)

The Resource Description Framework (RDF) [4, 5] is a framework for representing information in the Web. RDF graphs are sets of "subject-predicate-object triples" (fig. 2), where the elements may be IRIs[1], blank nodes, or datatyped literals. They are used to express descriptions of resources.

RDF datasets are used to organize collections of RDF graphs, are comprised of a default graph and zero or more named graphs.

[1]Internationalized Resource Identifier

The core structure of the abstract syntax is a set of triples, each consisting of a subject, a predicate and an object. A set of such triples is called an RDF graph. An RDF graph can be visualized as a node and directed-arc diagram, in which each triple is represented as a node-arc-node link.

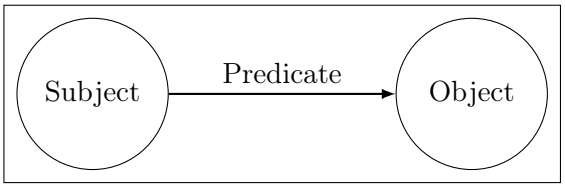


Figure 2: RDF SPO

1.4 SPARQL

SPARQL is the standard query language for RDF data[6]. SPARQL contains capabilities for querying required and optional graph patterns along with their conjunctions and disjunctions. SPARQL also supports aggregation, subqueries, negation, creating values by expressions, extensible value testing, and constraining queries by source RDF graph. The results of SPARQL queries can be result sets or RDF graphs.[7].

SPARQL query consists of a `SELECT` clause and a `WHERE` clause. The `SELECT` clause specifies a sequence of variables, separated by spaces. In SPARQL, variables start with a question mark. Each element of a triple can be an IRI, a variable or a literal. The result of the query is a table with k columns, where k is the number of variables in the `SELECT` clause. A row of the result table corresponds to an assignment of each variable of the `WHERE` clause to an IRI or literal. Each assignment must match in the sense that each triple of the `WHERE` clause exists in the knowledge graph, when plugging in the entity or literal for each variable. The keyword `FILTER` restricts the result table to rows that fulfill the specified expression. [3]

1.5 QLever

QLever (pronounced "Clever") is a SPARQL engine that can efficiently index and query very large knowledge graphs with over 100 billion triples on a single standard PC or server. In particular, QLever is fast for queries that involve large intermediate or final results, which are notoriously hard for engines like Blazegraph or Virtuoso. QLever also supports search in text associated with the knowledge base, as well as SPARQL autocompletion [8, 9].

The following is an example from QLever's (section 1.5) demos showcasing a SPARQL query (listing 1.1) that uses wikidata to list the birth place of people with a particular name using QLever [8, 9].

```
1  PREFIX wd: <http://www.wikidata.org/entity/>
2  PREFIX wdt: <http://www.wikidata.org/prop/direct/>
3  PREFIX p: <http://www.wikidata.org/prop/>
4  PREFIX psn: <http://www.wikidata.org/prop/statement/value-normalized/>
5  PREFIX wikibase: <http://wikiba.se/ontology#>
6  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
7  SELECT ?person ?person_label ?place_label ?coord WHERE {
8    ?person wdt:P31 wd:Q5 .
9    ?person wdt:P735/rdfs:label "Patrick"@en .
10   ?person wdt:P19 ?place .
11   ?place wdt:P625 ?coord .
12   ?person rdfs:label ?person_label .
13   ?place rdfs:label ?place_label .
14   FILTER (LANG(?person_label) = "en") .
15   FILTER (LANG(?place_label) = "en") .
16 }
```

Listing 1.1: Birthplaces of people named Patrick

wdt:P31 (Instance)[10] that class of which this subject is a particular example and member; different from P279 (subclass of); for example: K2 is an instance of

[2] showing 10 out of 4577 total results

?person	?person_label	?place_label	?coord
Q102116670	Patrick Kenney	Boston	POINT(-71.057778 42.360278)
Q17626715	Patrick Grant	Boston	POINT(-71.057778 42.360278)
Q105923793	Patrick H. O'Connor	Boston	POINT(-71.057778 42.360278)
Q110138990	Edward P. Barry, Jr.	Boston	POINT(-71.057778 42.360278)
Q1267622	Patrick Ewing, Jr.	Boston	POINT(-71.057778 42.360278)
Q105731759	Patrick E. Murray	Boston	POINT(-71.057778 42.360278)
Q18912678	Maurice Patrick Foley	Boston	POINT(-71.057778 42.360278)
Q46978642	Patrick O'Brien	Boston	POINT(-71.057778 42.360278)
Q41449445	Patrick Sweeney	Boston	POINT(-71.057778 42.360278)
Q955405	Patrick Joseph Kennedy	Boston	POINT(-71.057778 42.360278)
...	...	...	...

Table 1: Truncated Result[2]of Query. listing 1.1

mountain; volcano is a subclass of mountain (and an instance of volcanic landform)

wdt:Q5 (Human)[11] any member of Homo sapiens, unique extant species of the genus Homo, from embryo to adult.

wdt:P735 (Given Name)[12] first name or another given name of this person; values used with the property should not link disambiguations nor family names.

wdt:P19 (Birth Place)[13] most specific known birth location of a person, animal or fictional character.

wdt:P625 (Coordinate Location)[14] geocoordinates of the subject.

2 Background

2.1 Query Planner

In spite of numerous advances in database management systems since the late 70s, the high-level concepts discussed in Selinger et al. [15] still being actively implemented and used as a reasonable starting point. For the past 40 years DBMSs have been heavily influenced by the System R model as described in *"Access Path Selection in a Relational Database Management System*[1]*"* . System R was an experimental database management system developed to carry out research on the relational model of data and was designed and built by members of the IBM San Jose Research Laboratory [15]. System R's model is still popular and widely mimicked, since it defined a systematic framework where the plan space enumeration (chapter 5) is defined independent of cost model (chapter 4) and the set of heuristics used in the optimization algorithm [16, p. 128]

Obtaining the result of any query requires going through four phases: *parsing*, *optimization*, *code generation* and *execution*. We are only concerned with the *optimization* aspect. SPARQL statements (section 1.4) are designed to be declarative and don't require the user to specify anything about the access path to be used for tuple retrieval, nor specify in what order joins are performed [15].

Out of all the possible permutations that a particular query with multiple joins can be executed, the planner picks the order that minimizes the total access cost

[1] Bible of Optimization

of executing the entire statement. the total access cost depends of a multitude of factors which are explored in details (chapter 4).

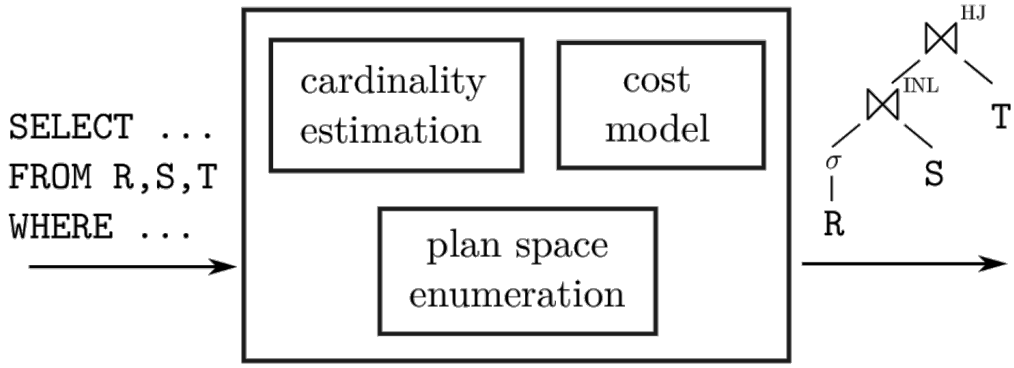
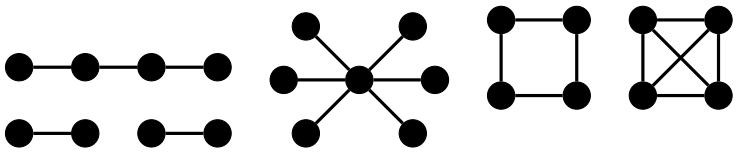


Figure 3: Traditional Query Planner Architecture [17]

2.2 Query Graph

A Query Graph is helpful representation of a given query. Modeling relations as nodes and the predicates as edges opens the door for graph algorithms to be utilized in the domain of query planners.

The complexity dependant on the query shape [18] dp-wise, chains (fig. 4a) are the easiest, cliques (fig. 4d) are the hardest [19, p. 196]



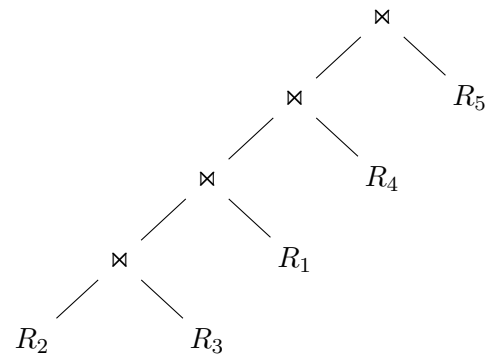
(a) linear/chain **(b)** star **(c)** cycle **(d)** clique

Figure 4: Query Graph Shapes [20, 3.2]

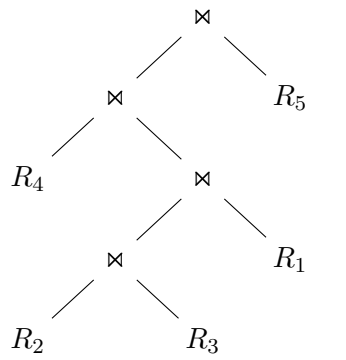
2.3 Join Tree

A join tree [2] [19, p. 76] is a binary tree whose leaf nodes are the relations and whose inner nodes are joins[3].

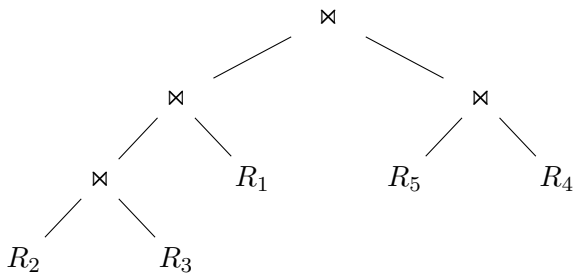
Join trees are often classified [20] into *left-deep trees* (fig. 5a) where every join has one of the relations R_i as its right input. *right-deep trees* where every join has one of the relations R_i as its left input. *zig-zag trees* (fig. 5b) where at least one input of every join is a relation R_i. *bushy trees* (fig. 5c) where no restrictions apply. the first three are summarized as linear trees [19].



(a) Sample left-deep tree
$((((R_2 \bowtie R_3) \bowtie R_1) \bowtie R_4) \bowtie R_5)$



(b) Sample zig-zag tree
$((R_4 \bowtie ((R_2 \bowtie R_3) \bowtie R_1)) \bowtie R_5)$



(c) Sample bushy tree
$(((R_2 \bowtie R_3) \bowtie R_1) \bowtie (R_5 \bowtie R_4))$

Figure 5: Linear Trees Structure

[2] sometimes are referred to as *Binary Join Processing Trees (BJTs)* [21, p. 9]
[3] possibly cross products

2.4 Rule-Based Optimization

The Rule-Based approach popularized by DBMSs like SQUIRAL [22], Starburst [23], EXODUS [24] relay on a predefined set of rules to repeatedly simplify the given expression until no more transformations can be applied.

Most of the database common-wisdom rules such as filtering tuples as early as possible (predicate pushdown) or delaying cross product as late as possible are utilized. Query planners restrict the space further by postponing cross-products as late as possible. The intuition is that cross-products are expensive and result in large intermediate results [21, p. 9].

Ensuring the correctness and equivalence of the output (two expression are said to be equivalent if the generate the same output) applying rules requires solid theoretical guarantees that gets more and more difficult as rules become more and more complex.

$$R_1 \bowtie R_2 = R_2 \bowtie R_1 \qquad \text{joining is commutative} \qquad (1)$$

$$(R_1 \bowtie R_2) \bowtie R_3 = R_1 \bowtie (R_2 \bowtie R_3) \qquad \text{joining is associative} \qquad (2)$$

For example in eq. (3), a predicate with a few conjunctive clauses can be broken down in separate individual operators and still produce the same output.

$$\sigma_{p_1 \wedge p_2 \wedge p_3}(R) \equiv \sigma_{p_1}(\sigma_{p_2}(\sigma_{p_3}(R))) \qquad (3)$$

2.5 Cost-Based Optimization

The Cost-Based approach popularized by Selinger et al. [15, 20] to evaluate potential candidate plans is the to aggregate all the sources of delay that a plan execution might encounter. All delay sources can be boiled down to CPU cost and I/O cost:

$$C = C_{I/O} + wC_{CPU}$$

where w is an optional weighting factor that can be adjusted or completely discarded when the system is not CPU bound. weight of 0.5 implies that 50% of the CPU time spent on a given plan will run in parallel with the I/O time. However, if we assume the system is capable of total concurrency is assumed, the cost function can be formulated as:

$$C = \texttt{max}(C_{I/O}, C_{CPU})$$

The $C_{I/O}$ and C_{CPU} of a given plan is the total sum of the costs of each operator:

$$C_{I/O} = \sum_{operator \in plan} Cf_{I/O}(operator) \tag{4}$$

$$C_{CPU} = \sum_{operator \in plan} Cf_{CPU}(operator) \tag{5}$$

The most influential factor when I/O cost of a given operator is the number of page read (page fetches), while for CPU cost it's the number of calls "Storage Interface" (RSI) the estimate the number of tuples. In case of iterator-based implementation, it is the number of invocations of a `next()` procedure to get the next tuple for processing.

3 Cardinality Estimation

Obtaining an accurate estimate for a given relation's cardinality is critical for generate a good plan [20, 24.2.6]. As small errors in calculating each relation's cardinality will propagate during joining operations. for example, joining 5 relation $R_1 ... R_5$ and each relation with an erroneous estimate by a factor of 2; the cardinality estimate of $R_1 \bowtie R_2 ... \bowtie R_5$ will drift by a factor of 32.

Cardinality estimation boils down to counting the number of distinct in a multiset[1]. Once data has grown so much and it becomes infeasible to keep track of the exact count (due to the growing memory requirements), the proper course of action is to use a probabilistic cardinality estimator such as *Flajolet-Martin*, *BJKST*, *HyperLogLog* (or one of their many friends) since they require substantially less Memory. Harmouch et al. have evaluated the accuracy, runtime, and memory consumption of each of them [25].

Despite not being the main focus of this thesis, it's still worth covering the preliminaries of cardinality estimating and selectivity estimating; as even exhausting the whole plan space with a bad estimate leads to wrong costs and wrong costs lead to bad plans [17].

In most popular DBMSs [26], cardinality estimates operate under three assumptions, *Uniformity*, *Independence* and *Inclusion*. these are traditional assumptions[2] regard-

[1] Count-distinct problem
[2] approximating reality

ing the uniformity of the distribution of values and the independence with respect to each other [15, 16].

1. Assuming *Uniformity* means that all values have the same number of tuples, distinct values are evenly spaced and they have the same frequency.

2. Assuming *Independence* of predicates means predicates on attributes in the same table are independent, so when calculating selectivity of a conjunctive clause is simply multiplying the selectivity of each predicate.

3. Assuming *Inclusion* means the domain of the join keys overlap such that the smaller domain have matches in the larger domain [17].

3.1 Selectivity Estimation

Selectivity of predicates[3] is a value in the interval [0, 1] and defined as the fraction of entries in a data set or relation that satisfies some specified predicate [27, 2.1]. eq. (6) gives approximate selectivity factors for different kinds of predicates [15, p. 26].

$$f = \begin{cases} \frac{1}{\texttt{card(?property)}} & \texttt{?property = ?value} \\ \frac{1}{\texttt{max(card(?property1), card(?property2))}} & \texttt{?property1 = ?property2} \\ \frac{\texttt{max(?property) - value}}{\texttt{min(?property) - value}} & \texttt{?property > ?value} \\ \frac{\texttt{value2 - value1}}{\texttt{max(?property) - min(?property)}} & \texttt{?value1 < ?property < ?value2} \\ \texttt{sz(values)} * f(\texttt{?value1}) & \texttt{?property IN [?value1, ...]} \\ f(\texttt{pred1}) + f(\texttt{pred2}) - f(\texttt{pred1}) * f(\texttt{pred2}) & \texttt{pred1 OR pred2} \\ f(\texttt{pred1}) * f(\texttt{pred2}) & \texttt{pred1 AND pred2} \\ 1 - f(\texttt{pred}) & \texttt{NOT pred} \end{cases} \tag{6}$$

[3] also known as *multiplicity factor, filter factor, reduction factor or filter selectivity*

The selectivity $f_{i,j}$, with respect to the join of R_i and R_j, is defined to be the expected fraction of tuple pairs from R_i and R_j that will join [16].

$$f_{i,j} = \frac{\text{expected no. of tuples in the result of joining } R_i \text{ and } R_j}{\text{no. of tuples in } R_i \star \text{ no tuples of } R_j} = \frac{|R_i \bowtie_{p_{i,j}} R_j|}{|R_i \times R_j|} \quad (7)$$

Estimating selectivity is the number of join result tuples divided by the cartesian product of R_i & R_j. [16, 19, 20, p. 128,p. 76,p. 34]. If the $f_{i,j}$ is 0.01, then only 1% of cartesian product's tuples left after apply the join predicate $p_{i,j}$.

$$|R_i \bowtie_{p_{i,j}} R_j| = f_{i,j}|R_i||R_j|$$

3.2 Histogram-Based

Poosala et al. [28] define a histogram on attribute X as partitioning the data distribution $\mathcal{T}$ into β (≥ 1) mutually disjoint subsets called *buckets* and approximating the frequencies and values in each bucket in some common fashion.

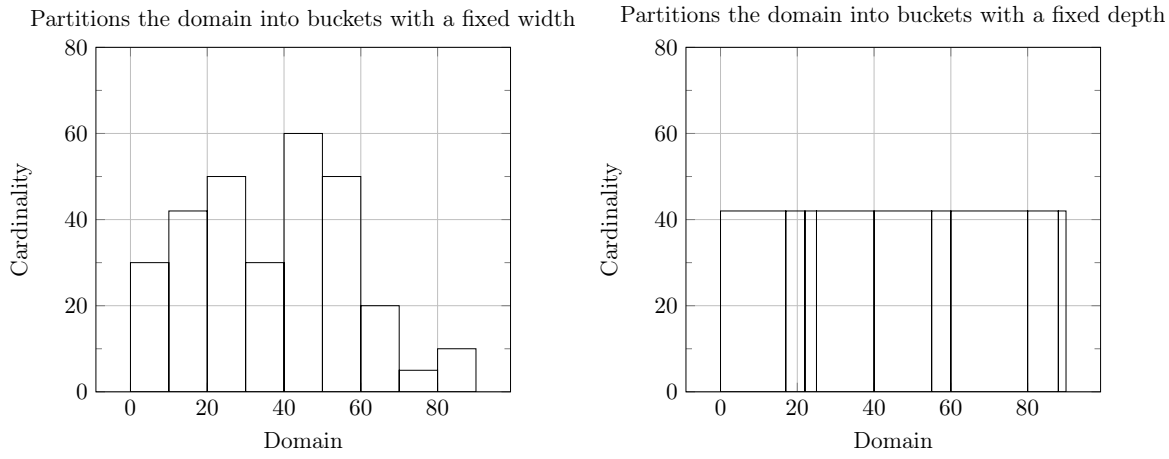
System R optimizer [15] uses trivial statistics [28, 4.1], such as the minimum and maximum values (see table 2) in a each column to estimate selectivity factors. Using such simple statistics will produce good selectivity estimates when the values are uniformly distributed [29].

Equi-Width histograms [28, 4.2] (fig. 6a) group contiguous ranges of attribute values into buckets, and the sum of the spreads in each bucket[4] is approximately equal to $1/\beta$ times the maximum minus the minimum value that appears in $\mathcal{V}$.

[4] i.e., the maximum minus the minimum value in the bucket

Table 2: `pg_stats` Columns [30, 31, 52.27]

Column	Type	Description
`schemaname`	`name`	*Name of schema containing table*
`tablename`	`name`	*Name of table*
`attname`	`name`	*Name of column described by current row*
`inherited`	`bool`	*If true, this row includes values from child tables*
`null_frac`	`float4`	*Fraction of column entries that are null*
`avg_width`	`int4`	*Average width in bytes of column's entries*
`n_distinct`	`float4`	*If greater than zero, the estimated number of distinct values in the column. If less than zero, the negative of the number of distinct values divided by the number of rows.*
`most_common_vals`	`anyarray`	*A list of the most common values in the column.*
`most_common_freqs`	`float4[]`	*A list of the frequencies of the most common values, i.e., number of occurrences of each divided by total number of rows.*
`histogram_bounds`	`anyarray`	*A list of values that divide the column's values into groups of approximately equal population.*
`correlation`	`float4`	*Statistical correlation between physical row ordering and logical ordering of the column values.*
`most_common_elems`	`anyarray`	*A list of non-null element values most often appearing within values of the column.*
`most_common_elem_freqs`	`float4[]`	*A list of the frequencies of the most common element values, i.e., the fraction of rows containing at least one instance of the given value.*
`elem_count_histogram`	`float4[]`	*A histogram of the counts of distinct non-null element values within the values of the column, then, the average number of distinct non-null elements.*
`range_length_histogram`	`anyarray`	*A histogram of the lengths of non-empty and non-null range values of a range type column.*
`range_empty_frac`	`float4`	*Fraction of column entries whose values are empty ranges.*
`range_bounds_histogram`	`anyarray`	*A histogram of lower and upper bounds of non-empty and non-null range values.*



(a) Equi-Width Histogram **(b)** Equi-Depth Histogram

Figure 6: Building Histograms [19, p. 569-573]

Equi-Depth histograms [29, 28, 4.3] (fig. 6b) have the sum of the frequencies in each bucket be equal rather than the sum of the spreads, only the boundaries of buckets needs to be stored[5]. (table 2)

3.3 Sampling-Based

Sampling-Based estimation can be a viable alternative to Histogram-Based estimation (section 3.2) due to it's ability to detect correlations between relation's attributes and nonuniform data [32, 33], Unlike histogram-based methods, Sampling don't require storing and maintaining detailed statistics about the base data in the database's catalog [34, 35]

Lipton and Naughton proposed "Adaptive Random Sampling" [34] in the early 90s; an algorithm to estimate the size of a general given query ($\mathcal{Q}$) by partitioning the query into disjoint subsets ($\mathcal{Q}_1, \mathcal{Q}_2, \ldots, \mathcal{Q}_n$), then count the size of randomly chosen subsets. The running is directly proportional to size of the sample and the cost it take to compute the samples. this estimation algorithm's termination condition

[5] `pg_catalog.pg_stats` in the case of PostgreSQL

expressed in terms of the size of the sum of the samples taken, rather than in terms of the number of samples; giving it an adaptive flavor. If the samples are large; fewer will be taken. if the samples are small; more will be taken [35].

3.4 Machine Learning-Based

All machine leaning estimators have in common that they do not consume a query, e.g., a SQL string, directly. Instead, a numerical representation of a query, called feature vector, is consumed. A function that maps a query to its feature vector is called query featurization technique (QFT) [36, 4.4].

Kipf et al. featurize queries into different sets and learn their cardinalities with a specific Multi Set Convolutional Network (MSCN) architecture [37] [36, 4.4].

4 Cost Model

4.1 ASI Property

One of the important ideas in the theory of sequencing and scheduling is the method of adjacent pairwise job interchange. This method compares the costs of two sequences which differ only by interchanging a pair of adjacent jobs [1, p. 217].

Monma et al. describes [1, 38] any cost function C to have the *Adjacent Sequence Interchange* property, if and only if there exists a cost-benefit ration function `rank(s)` and function $T(S)$ for sequence S

$$\texttt{rank}(S) = \frac{T(S) - 1}{C(S)} \tag{8}$$

such that for all sequences a, b and all non-empty sequences v, u the following hold:

$$C(a, u, v, b) \leq C(a, v, u, b) \iff \texttt{rank(u)} \leq \texttt{rank(v)} \tag{9}$$

if $auvb$ and $avub$ satisfy the precedence constraints imposed by a given precedence graph [39, A.].

`rank` function measures the increase in the intermediate result per unit differential cost of doing join [16, 4.2].

4.2 Cost Function: Cout

Cluet et al. proposed cost function C_{out} that computes the sum of the sizes of the intermediate results after join operations, which is very suitable for our purposes since it is consistent with the common database wisdom that minimizing the intermediate results after join is a good heuristic.

Moreover, they showed that C_{out} has the ASI property (section 4.1) [38, Observation 7] which will be necessary when using some join-ordering algorithms like (algorithm 5 in section 5.2).

Under the the cost of writing the intermediate results to disk outweighs any CPU cost, C_{out} can be defined as:

$$C_{out}(R_i \bowtie R_j) := |R_i||R_j|f_{i,j}$$

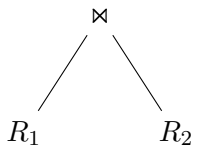
[19, p. 77] Given a join tree T, the result of cardinality $|T|$ can be computed recursively as:

$$|T| = \begin{cases} |R_i| & \text{if } T \text{ is a leaf } R_i \\ (\prod_{R_i \in T_1, R_j \in T_2} f_{i,j})|T_1||T_2| & \text{if } T = T_1 \bowtie T_2 \end{cases} \tag{10}$$

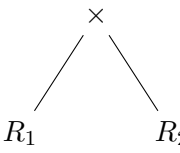
[19, p. 79] [38] Given a join tree T (section 2.3), the cost function C_{out}

$$C_{out}(T) = \begin{cases} 0 & \text{if } T \text{ is a leaf of } R_i \\ |T| + C_{out}(T_1) + C_{out}(T_2) & \text{if } T = T_1 \bowtie T_2 \end{cases} \tag{11}$$

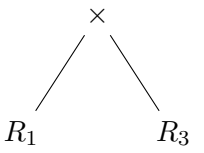
Figure 7: C_{out} where $|R_1| = 1000, |R_2| = 2, |R_3| = 2$, $f_{1,2} = 0.1, f_{1,3} = 0.1$ [19, p. 83]



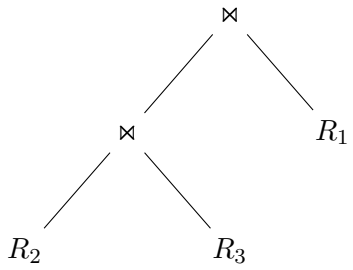
(a) $C_{out}(R_1 \bowtie R_2) = 1000 * 2 * 0.1 = 200$



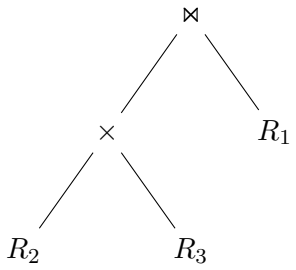
(b) $C_{out}(R_1 \times R_2) = 2 * 2 = 4$



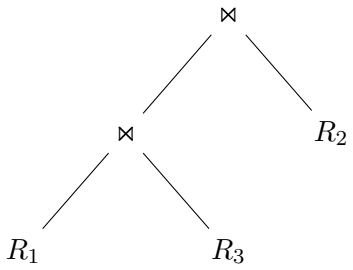
(c) $C_{out}(R_1 \bowtie R_3) = 1000 * 2 * 0.1 = 200$



(d) $C_{out}((R_1 \bowtie R_2) \bowtie R_3) = 1000*2*0.1+1000*2*0.1*2*0.1 = 240$

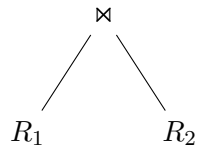


(e) $C_{out}((R_2 \times R_3) \bowtie R_1) = 2*2+2*2*1000*0.1*0.1 = 44$

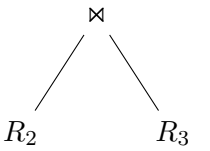


(f) $C_{out}((R_1 \bowtie R_3) \bowtie R_2) = 1000*2*0.1+1000*2*0.1*2*0.1 = 240$

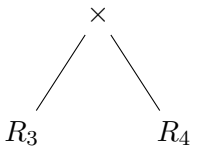
Figure 8: C_{out} where $|R_1| = 10, |R_2| = 20, |R_3| = 20, |R_4| = 10$, $f_{1,2} = 0.01, f_{2,3} = 0.5, f_{3,4} = 0.01$[19, p. 84]



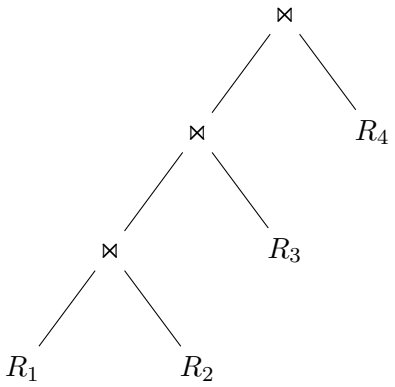
(a) $C_{out}(R_1 \bowtie R_2) = 10 * 20 * 0.01 = 2$



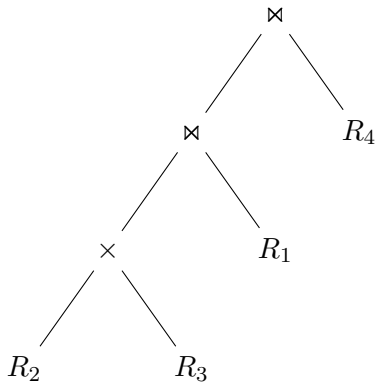
(b) $C_{out}(R_2 \bowtie R_3) = 20 * 20 * 0.5 = 200$



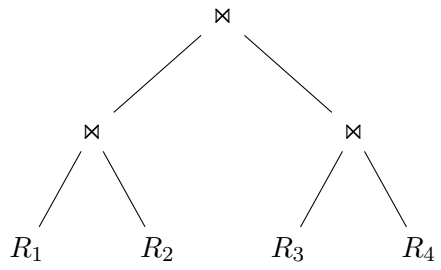
(c) $C_{out}(R_3 \bowtie R_4) = 20 * 10 * 0.01 = 2$



(d) $C_{out}(((R_1 \bowtie R_2) \bowtie R_3) \bowtie R_4) = 10 * 20 * 0.01 + 10 * 20 * 0.01 * 20 * 0.5 + 10 * 20 * 0.01 * 20 * 0.5 * 10 * 0.01 = 24$



(e) $C_{out}(((R_2 \times R_3) \bowtie R_1) \bowtie R_4) = 20 * 20 * 0.5 + 20 * 20 * 0.5 * 10 * 0.01 + 20 * 20 * 0.5 * 10 * 0.01 * 10 * 0.01 = 222$



(f) $C_{out}((R_1 \bowtie R_2) \bowtie (R_3 \bowtie R_4)) = 10 * 20 * 0.01 + 20 * 10 * 0.01 + 2 = 6$

5 Plan Enumeration

5.1 DP-Based

Bellman defines Principle of Optimality [40, p. 83] as optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

The idea of dynamic programming applied to the generation of optimal join trees is to generate optimal join trees for subsets of $R_1, \ldots, R_n$ in a bottom-up fashion. Optimal join trees for subsets of size one (single relations) are generated. From these, optimal join trees of size two, three and so on until n are generated [20, p. 61].

The dynamic programming algorithms are often the core of commercial DBMSs query planners. But solely relying on DP-based approaches is not recommended as runtime increases exponentially with respect to the number of relations involved [20, p. 69]. for example, PostgreSQL switches from DP-based approach to heuristic-based approach for queries that involve more than 12 relations [41].

In this section, we will explore some of the most common DP-based algorithms with accompanying pseudocode.

Size-Driven Enumeration

Selinger et al. introduced the idea of using dynamic programming for finding optimal bushy plans `DPsize` [15, 42, 43] by synthesizing plans of increasing sizes. `DPsize` (algorithm 1) runs in exponential time complexity $\mathcal{O}(2^N)$ with an exponential space requirement [21].

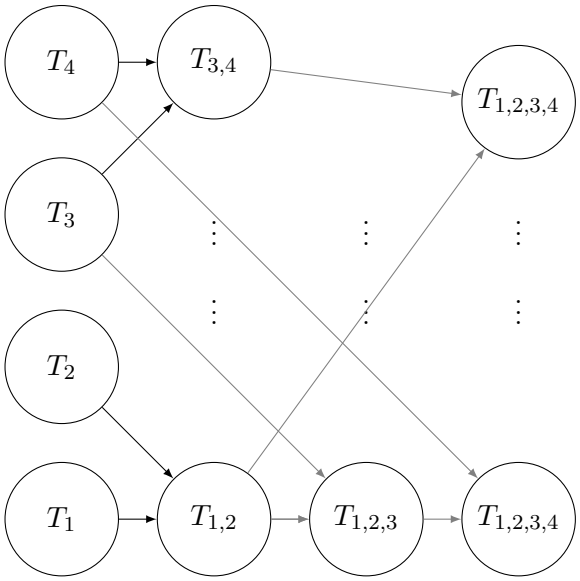
Shown in Algorithm 1, sets of relations contained in s_l and s_r do not overlap, and ensures the existence a join predicate connecting a relation s_l with a relation in s_r, while dp associates each set of relations the best plan found so far [20, p. 70].

Algorithm 1 DPsize [44, p. 540]

Input: connected query graph Q with n-relations ($R \leftarrow \{R_0, \ldots, R_{n-1}\}$)
Output: optimal bushy join tree without cross products

```
1:  foreach R_i ∈ R do
2:      dp[1 << i] ← R_i                                   ▷ init 2^n DPTable
3:  end for
4:  foreach s ∈ {2, ..., n} do                             ▷ size of plan
5:      foreach s_l ∈ {1, ..., s − 1} do                   ▷ size of left subplan
6:          s_r ← s − s_l                                  ▷ size of right subplan
7:          foreach S_l ⊂ R do                             ▷ all plans containing s_l relations
8:              foreach S_r ⊂ R do                         ▷ all plans containing s_r relations
9:                  if S_l ∩ S_r ≠ ∅ continue              ▷ no overlap
10:                 if S_l not connected to S_r continue   ▷ existence of join predicate
11:                 p ← dp[S_l] ⋈ dp[S_r]                  ▷ current plan
12:                 if cost(p) < cost(dp[S_l ∪ S_r]) then  ▷ relations contained in p
13:                     dp[S_l ∪ S_r] ← p
14:                 end if
15:             end for
16:         end for
17:     end for
18: end for
19: return dp[{R_0, ..., R_{n−1}}]                         ▷ optimal plan
```

Figure 9: DPsize Algorithm Execution [45, p. 55]



Subset-Driven Enumeration[1]

Vance et al. improved on (algorithm 1) by relaying on bit vector representation of relations set giving us `DPsub` [46, 42, 43] (algorithm 3). It relies on Efficient Subset Generation [46, 47, 4.1] and the fact that the increment by one operation is very simple operation and can be used to generate the powerset of a given set as seen in (algorithm 3). Each relation is represented by the i-th bit in the bitvector (table 3).

Algorithm 2 Efficient Subset Generation [19, p. 159]

1: $S_1 \leftarrow S\&(-S)$
2: **do**
3: $\quad S_2 \leftarrow S - S_1$
4: $\quad \ldots$ ▷ do something with S_1, S_2
5: $\quad S_1 \leftarrow S\&(S_1 - S)$
6: **while** $S_1 \neq S$

As demonstrated by Moerkotte [42, 43] that `DPsize` (algorithm 1) is superior to `DPsub` for chain and cycle queries (figs. 4a and 4c). while, `DPsub` is superior to `DPsize` (algorithm 1) for star and clique queries (figs. 4b and 4d).

[1] also know by Counter-Driven Enumeration

Table 3: Generation in Integer Order[19, p. 156]

Int	Bin	Relations
0	000	$\{\}$
1	001	$\{R_1\}$
2	010	$\{R_2\}$
3	011	$\{R_1, R_2\}$
4	100	$\{R_3\}$
5	101	$\{R_1, R_3\}$
6	110	$\{R_2, R_3\}$
7	111	$\{R_1, R_2, R_3\}$

Algorithm 3 DPsub [42, p. 932]

Input: connected query graph Q with n-relations ($R \leftarrow \{R_0, \ldots, R_{n-1}\}$)
Output: optimal bushy join tree

1: **foreach** $R_i \in R$ **do**
2: $dp[1 << i] \leftarrow R_i$ ▷ init 2^n DPTable
3: **end for**
4: **foreach** $S \in \{1, \ldots, 2^n - 1\}$ **do**
5: **if** (not connected S) **continue**
6: **foreach** $S_1 \subset S$ **do**
7: $S_2 \leftarrow S \setminus S_1$
8: **if** $S_2 = \emptyset$ **continue**
9: **if** not connected S_1 **continue**
10: **if** not connected S_2 **continue**
11: **if** S_1 not connected to S_2 **continue** ▷ existence of join predicate
12: $p \leftarrow dp[S_1] \bowtie dp[S_2]$ ▷ current plan
13: **if** $\texttt{cost}(p) < \texttt{cost}(dp[S])$ **then**
14: $dp[S] \leftarrow p$
15: **end if**
16: **end for**
17: **end for**
18: **return** $dp[\{R_0, \ldots, R_{n-1}\}]$ ▷ optimal plan

5.2 IKKBZ

DP-based approaches alone (section 5.1) are not enough for modern DBMSs with regular consumer hardware. In this section we will relax the global optimality requirement for queries involving large number relations in exchange for a polynomial-time algorithm.

Ibaraki and Kameda got their inspiration from Monma et al. work in finding a general algorithm for sequencing problems with series-parallel precedence from the field of Scheduling & Operation Research.

Monma et al. [1, p. 216] original motivation is a tackling the problem of least cost fault detection, where in a system of consisting of n components is to be inspected sequentially by applying tests to each component until one fails or each component pass it's test.

Each component j has testing cost c_j and a probability $0 \leq q_j \leq 1$ of passing it's test. the probability that the i-th component in sequence s $\mathcal{Q}_i^s = q_{s(1)}q_{s(2)} \cdots q_{s(i-1)}$. the expected testing costs for a sequence s of length k is $\sum_{i=1}^{k} \mathcal{Q}_i^s c_{s(i)}$. the problem is to find feasible permutation Π which minimizes the expected cost:

$$\min_{\Pi \in F} \sum_{i=1}^{k} \mathcal{Q}_i^s c_{s(i)}$$

Recursive definition of the cost function f for $\sum \mathcal{Q}_i c_i$ is defined on all sequences as:

$$\begin{aligned} f(j) &= c_j && \text{for job } j \\ f(s,t) &= f(s) + q(s)f(t) && \text{for sequences } s,t \end{aligned}$$

where $q(s) = q_{s(1)}q_{s(2)} \cdots q_{s(k)}$ for sequence s of length k.

Krishnamurthy et al. improved[16] on Ibaraki et al. [1, 48] and propose $\mathcal{O}(n^2)$ algorithm 5 for finding the optimal left-deep tree (fig. 5a) for an acyclic[2] query graph (inner joins only) where in is the number of relations in the query.

In case of a query graph contains cycles, compute the minimum spanning tree[3] using algorithm 4 that minimize the product of all selectivities. The intuition behind that is a high selectivity impact over choosing the order [16, Observation 2].

Algorithm 4 Kruskal's Algorithm [49, 23.2]

Require: $G(V, E), w$
1: $A = \emptyset$
2: **foreach** $v \in G.V$ **do**
3: `MAKE-SET`(v)
4: **end for**
▷ sort the edges of $G.E$ into non-decreasing order by weight w
5: **foreach** $(u, v) \in G.E$ **do**
6: **if** `FIND-SET`$(u) \neq$ `FIND-SET`(v) **then**
7: $A = A \cup \{(u, v)\}$
8: `UNION`(u, v)
9: **end if**
10: **end for**
11: **return** A

Once we have an acyclic graph, pick the first relation and construct a precedence tree where the relation is the root of the tree and every other relation is pointing outwards as seen in fig. 11.a. This indicates which relations have to be joined first before other joins become feasible [41, p. 7].

Then, *Normalize* and *Merge Chains* repeatedly based on their rank for every subtree[4]. `rank`$(R_6) \leq$ `rank`(R_7) then R_6 should be joined before R_7.

The intuition behind *Normalization* is that if the direction stated in the precedence tree and the `rank` function disagree ($R_6 \rightarrow R_7$ but `rank`$(R_6) \leq$ `rank`(R_7)), this constitute a contradiction that can be resolved by combining these 2 relations (R_6, R_7)

[2] non-recursive
[3] or any other algorithm, i.e. Prim's
[4] sometimes referred to as wedge [48, p. 496]

into a single relation R_6R_7. This new compound relation is treated as a single unit when merging chains (fig. 11.b, fig. 11.c).

The intuition behind *merging chains* is due to the absence of any restrictions between the member relations of 2 (or more) parallel chains under the subtree, we can merge them into a single chain with sorted ranking (from smallest to largest).

Finally, the intuition behind *Denormalization* is after successive *Normalization* and *Merging Chains*, we are left with a single chain where we unpack any compound relations created from any normalization procedure in order to end up with the single relation that can be part of final join tree.

The aforementioned process is repeated[5] for every relation as the precedence tree root the tree with the lowest cost is considered the optimal left-deep tree of the given query. The total cost of the spanning tree is defined as the product of all the selectivities and riot the summation as it is commonly stated for the minimum cost spanning tree problem [16, p. 136].

For optimizing non-recursive queries, an polynomial $\mathcal{O}(N^2)$ heuristic search algorithm (algorithm 5). The theory [48, 16] is based requires that the cost functions have a certain form [21].

Recursive definition of the cost function [38, p. 60][19, p. 112][20, Definition 3.2.1]:

$$\begin{aligned} &C_H(\epsilon) = 0 \\ &C_H(R_i) = 0 && \text{if } R_i \text{ is the root} \\ &C_H(R_i) = h_i(n_i) && \text{if } R_i \text{ is not the root} \\ &C_H(S_1S_2) = C_H(S_1) + T(S_1) * C_H(S_2) \end{aligned} \tag{12}$$

$$\begin{aligned} T(\epsilon) &= 1 \\ T(S) &= \prod_{R_i \in S} s_i n_i \end{aligned} \tag{13}$$

[5] trivially parallelizable with `std::transform_reduce` & policy `std::execution::par_unseq`

Algorithm 5 IKKBZ [20, p. 54] [19, p. 120-123]

Require: an acyclic query graph G and an ASI cost function C_H

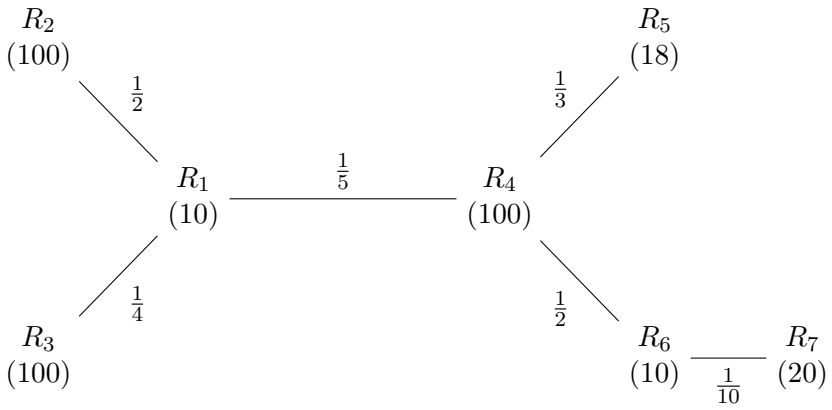
$S \leftarrow \emptyset$
foreach $R_i \in R$ **do** ▷ Consider each relation as starting relation
    $G_i \leftarrow$ Precedence graph derived from G rooted at R_i
    $S_i \leftarrow$ IKKBZ-Sub(G_i, C_H)
    $S \leftarrow S \cup \{S_i\}$
end for
return $\text{argmin}_{S_i \in S} C_H(S_i)$ ▷ optimal left-deep tree

procedure IKKBZ-SUB(G, C_H) ▷ precedence graph G, cost function C_H
    while G_i is not a chain **do**
        $r \leftarrow$ a subtree of G_i whose subtrees are chains
        IKKBZ-Normalize(r)
        ▷ merge chains under r according to rank function
    end while
    IKKBZ-Denormalize(G_i) ▷ optimal left-deep tree under G_i
end procedure

procedure IKKBZ-NORMALIZE(R) ▷ a subtree R of precedence graph G
    while $\exists r, c \in R, \texttt{rank}(r) > \texttt{rank}(c)$ **do**
        ▷ replace r, c by a compound relation r' that replace rc
    end while
end procedure ▷ normalized subtree

procedure IKKBZ-DENORMALIZE(G) ▷ precedence graph (compound relations)
    while $\exists r \in R : r$ is a compund relation **do**
        ▷ replace r sequence of relations it represent
    end while
end procedure ▷ denormalized precedence graph G

Figure 10: Sample Query Graph



(a) Query Graph

```
SELECT * FROM
R1, ..., R7 WHERE
R1.a = R2.a and
R1.b = R3.b and
R1.c = R4.c and
R4.d = R5.d and
R4.e = R6.e and
R6.f = R7.f
```

(b) SQL Query

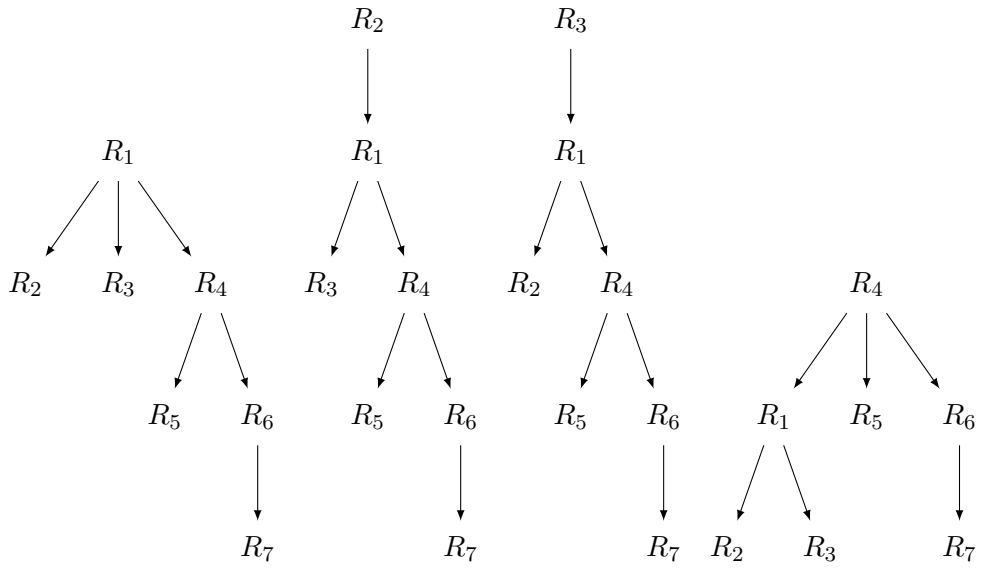
```
SELECT ?r1...?r7
WHERE {
  ?r1 a ?v1 .
  ?r1 c ?v2 .
  ?r1 c ?v3 .
  ?r2 b ?v2 .
  ?r3 c ?v3 .
  ?r4 d ?v4 .
  ?r5 e ?v5 .
  ?r6 f ?v6 .
  ?r7 g ?v7 .
  ?r3 c ?v3 .
}
```

(c) SPARQL Query

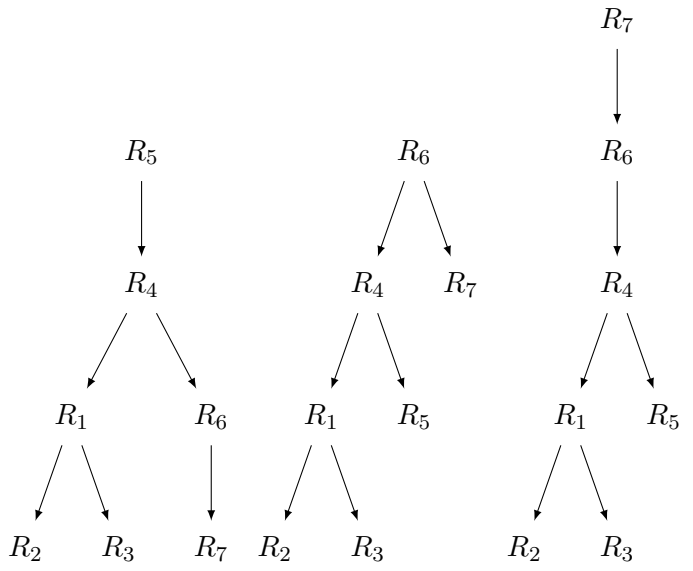
Table 4: Rank Computation for Figure 12

R	**n**	**s**	**C**	**T**	**rank**
R_1	10	0.20	2.00	2.00	0.50
R_2	100	0.50	50.00	50.00	0.98
R_3	100	0.25	25.00	25.00	0.96
R_4	100	0.20	20.00	20.00	0.95
R_5	18	0.33	6.00	6.00	0.83
R_6	10	0.50	5.00	5.00	0.80
R_7	20	0.10	2.00	2.00	0.50
R_6R_7	200	0.05	15.00	10.00	0.60
$R_4R_6R_7$	20000	0.01	320.00	200.00	0.62

Figure 11: Precedence Trees of Figure 10

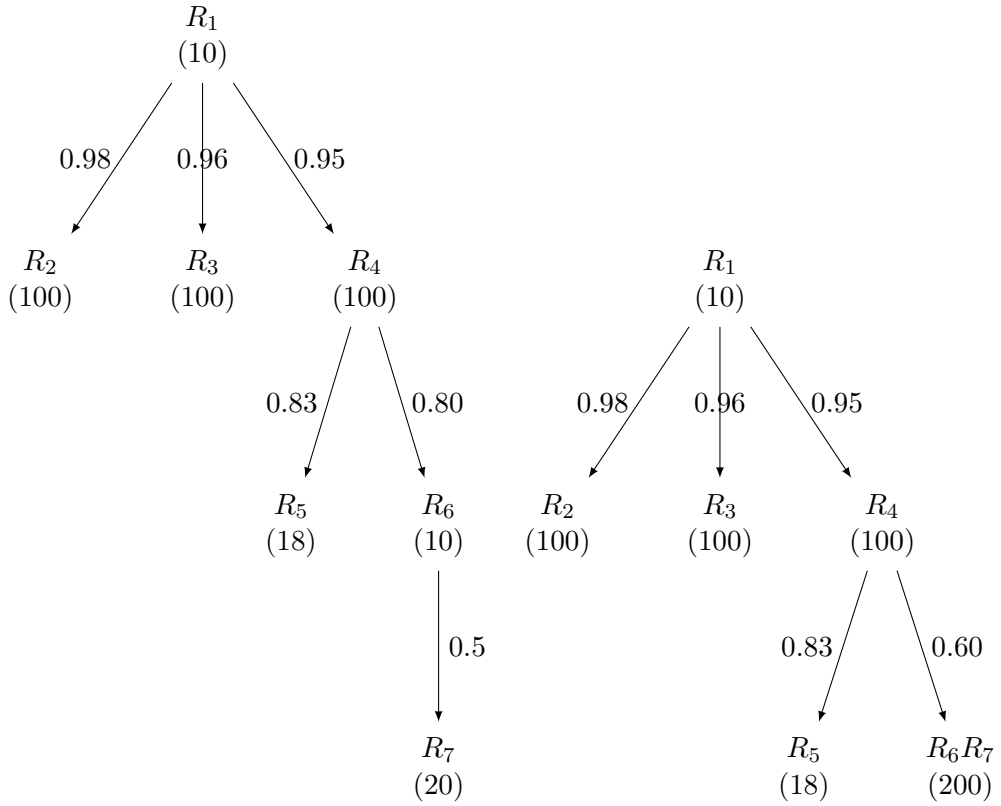


(a) rooted at R_1 **(b)** rooted at R_2 **(c)** rooted at R_3 **(d)** rooted at R_4



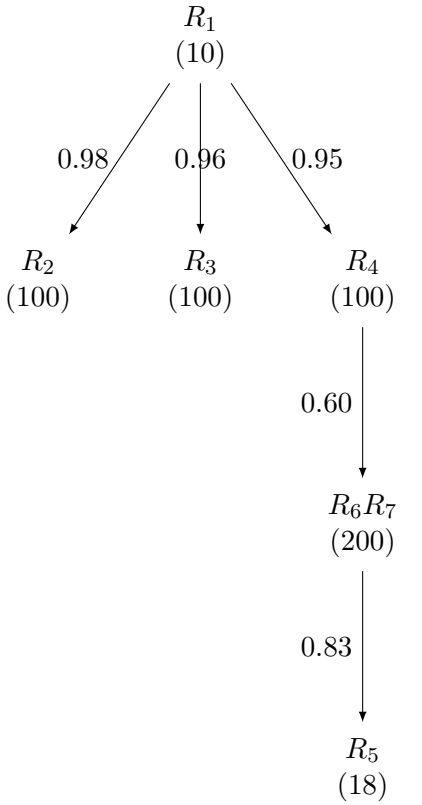
(e) rooted at R_5 **(f)** rooted at R_6 **(g)** rooted at R_7

Figure 12: IKKBZ Algorithm Execution [19, p. 124] for fig. 11.a

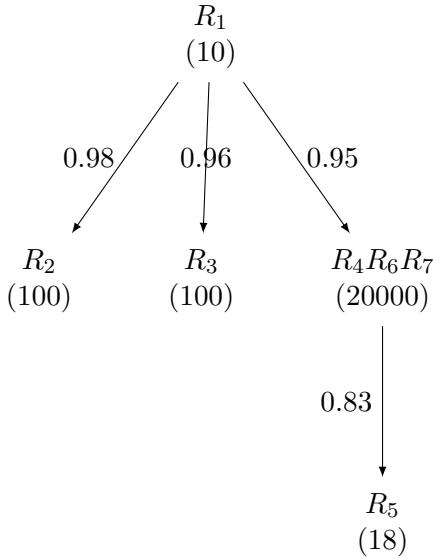


(a) precedence graph rooted at R_1

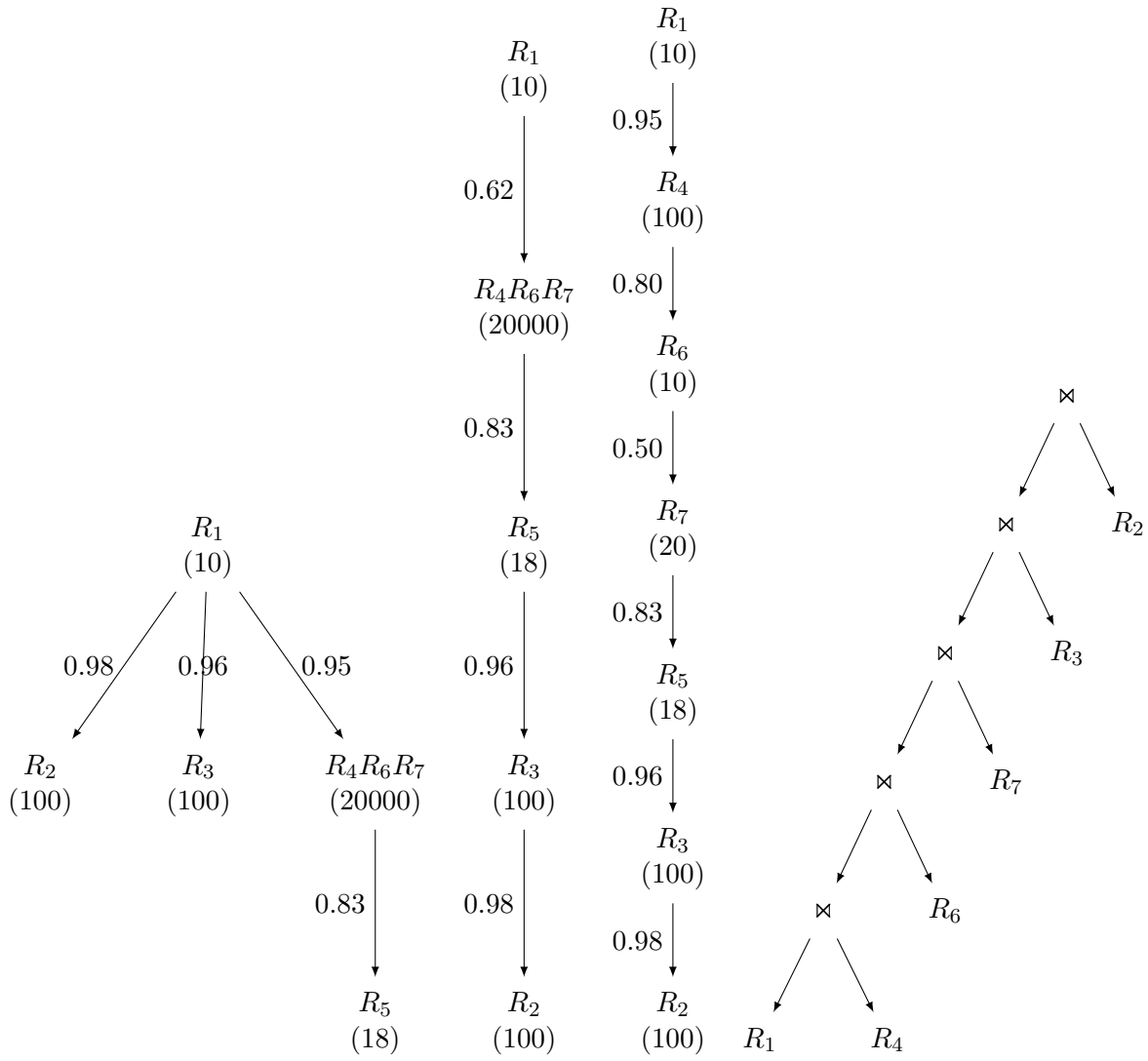
(b) $\text{rank}(R_6) > \text{rank}(R_7)$, but $R_6 \rightarrow R_7$



(c) merging subchains of R_4



(d) $\text{rank}(R_4) > \text{rank}(R_6, R_7)$, but $R_4 \to R_6R_7$



(e) rank(R_4) > rank(R_6, R_7), but $R_4 \to R_6R_7$

(f) merging subchains of R_1

(g) denormalized graph

(h) equivalent join tree

5.3 Linearized DP

As demonstrated in section 5.2, we can find optimal left-deep tree in $\mathcal{O}(n^2)$ time. Neumann et al. [50] showed that relative order can act as a really good starting point for one of the DP plan enumeration algorithms algorithms 1 and 3 to construct an optimal bushy tree for the given relative order [41, p. 2], such simplification reduces the amount fo plans considered by the DP-based enumeration algorithms. this process is known as State Space Linearization.

Algorithm 6 linDP [50, 4.2]

Require: $G(V, E), w, C_H$
1: $G' = \texttt{MST}(G, w)$ ▷ algorithm 4
2: $O = \texttt{IKKBZ}(G', C_H)$ ▷ algorithm 5
3: **foreach** $R_i \in R$ **do**
4: $dp[i, i] \leftarrow R_i$ ▷ init n^2 DPTable
5: **end for**
6: **foreach** $s \in \{2, \ldots, |O|\}$ **do**
7: **foreach** $i \in \{0, \ldots, |O| - s\}$ **do**
8: **foreach** $j \in \{1, \ldots, |O| - s - 1\}$ **do**
9: $L \leftarrow dp[i, i + j - 1]$ ▷ left subplan
10: $R \leftarrow dp[i + s, i + s - 1]$ ▷ right subplan
11: **if** L can join with R **then** ▷ existence of join predicate
12: $P \leftarrow L \bowtie R$ ▷ current plan
13: **if** $C(P) < C(dp[i, i + s - 1])$ **then**
14: $dp[i, i + s - 1] \leftarrow P$
15: **end if**
16: **end if**
17: **end for**
18: **end for**
19: **end for**
20: **return** $dp[0, |O| - 1]$ ▷ sub-optimal bushy tree

6 Conclusion

IKKBZ (section 5.2) has proven to be useful in any Query Planner for many reasons. it can be used on it's own to find an optimal-left deep plan in polynomial time which is good enough for most practical use-case[1], or act as a stepping stone for a DP-based enumeration algorithm that results in an bushy plan in a reasonable amount of time (section 5.1).

Although linearization (section 5.3) process no longer guarantee the optimality of the output plans and may result in suboptimal bushy plans, it still synthesize really good plans that are not far from the global optimal ones [50].

All the aforementioned algorithms heavily relay on accurate cardinality estimations (chapter 3) which get harder and harder as the number of joins increase and the amount data increase.

[1] Oracle for example, doesn't bother at all with bushy plans [17, 6.2]

7 Acknowledgments

I would like to thank Prof. Hannah Bast for the opportunity to work on this interesting and exciting topic. Also, I would like to thank Johannes Kalmbach for picking the title for the thesis.

Bibliography

[1] C. L. Monma and J. B. Sidney, "Sequencing with Series-Parallel Precedence Constraints," vol. 4, no. 3, pp. 215–224. [Online]. Available: https://www.jstor.org/stable/3689575

[2] Data dumps/Dumps sizes and growth - Meta. [Online]. Available: https://meta.wikimedia.org/wiki/Data_dumps/Dumps_sizes_and_growth

[3] H. Bast, J. Kalmbach, T. Klumpp, and C. Korzen, "Knowledge Graphs and Search," in *Information Retrieval*, 1st ed., O. Alonso and R. Baeza-Yates, Eds. ACM, pp. 231–281. [Online]. Available: https://dl.acm.org/doi/10.1145/3674127.3674134

[4] RDF - Semantic Web Standards. [Online]. Available: https://www.w3.org/RDF/

[5] RDF 1.2 Concepts and Abstract Syntax. [Online]. Available: https://www.w3.org/TR/rdf12-concepts/

[6] J. Kalmbach, "Efficient SPARQL Autocompletion on Large Knowledge Bases."

[7] SPARQL 1.2 Query Language. [Online]. Available: https://www.w3.org/TR/sparql12-query/

[8] H. Bast and B. Buchhold, "QLever: A Query Engine for Efficient SPARQL+Text Search," in *Proceedings of the 2017 ACM on Conference*

on Information and Knowledge Management. ACM, pp. 647–656. [Online]. Available: https://dl.acm.org/doi/10.1145/3132847.3132921

[9] "QLever," University of Freiburg: Algorithms and Data Structures Group. [Online]. Available: https://github.com/ad-freiburg/qlever

[10] Instance of. instance of (P31). [Online]. Available: https://www.wikidata.org/wiki/Property:P31

[11] Human. human (Q5). [Online]. Available: https://www.wikidata.org/wiki/Q5

[12] Given name. given name (P735). [Online]. Available: https://www.wikidata.org/wiki/Property:P735

[13] Place of birth. place of birth (P19). [Online]. Available: https://www.wikidata.org/wiki/Property:P19

[14] Coordinate location. [Online]. Available: https://www.wikidata.org/wiki/Property:P625

[15] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, ser. Sigmod '79. Association for Computing Machinery, pp. 23–34. [Online]. Available: https://doi.org/10.1145/582095.582099

[16] R. Krishnamurthy, H. Boral, and C. Zaniolo, "Optimization of Nonrecursive Queries," in *Proceedings of the 12th International Conference on Very Large Data Bases*, ser. VLDB '86. Morgan Kaufmann Publishers Inc., pp. 128–137.

[17] V. Leis, A. Gubichev, Atanas Mirchev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" vol. 9, no. 3, pp. 204–215.

[18] K. Ono and G. M. Lohman, "Measuring the Complexity of Join Enumeration in Query Optimization," in *Proceedings of the 16th International Conference on Very Large Data Bases*, ser. VLDB '90. Morgan Kaufmann Publishers Inc., pp. 314–325. [Online]. Available: https://www.cs.cmu.edu/~15721-f24/papers/Complexity_of_Join_Enumeration.pdf

[19] T. Neumann, "Query Optimization." [Online]. Available: https://db.in.tum.de/teaching/ws2324/queryopt/

[20] G. Moerkotte, *Building Query Compilers*. [Online]. Available: https://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf

[21] A. Swami and A. Gupta, "Optimization of large join queries," in *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '88. Association for Computing Machinery, pp. 8–17. [Online]. Available: https://dl.acm.org/doi/10.1145/50202.50203

[22] J. M. Smith and P. Y.-T. Chang, "Optimizing the performance of a relational algebra database interface," vol. 18, no. 10, pp. 568–579. [Online]. Available: https://dl.acm.org/doi/10.1145/361020.361025

[23] H. Pirahesh, J. M. Hellerstein, and W. Hasan, "Extensible/rule based query rewrite optimization in Starburst," in *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, ser. Sigmod '92. Association for Computing Machinery, pp. 39–48. [Online]. Available: https://doi.org/10.1145/130283.130294

[24] G. Graefe and D. J. DeWitt, "The EXODUS optimizer generator," in *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, pp. 160–172.

[25] H. Harmouch and F. Naumann, "Cardinality estimation: An experimental

survey," vol. 11, no. 4, pp. 499–512. [Online]. Available: https://dl.acm.org/doi/10.1145/3186728.3164145

[26] A. Shen. CS 186. CS 186. [Online]. Available: https://cs186berkeley.net/

[27] M. Müller, G. Moerkotte, and O. Kolb, "Improved selectivity estimation by combining knowledge from sampling and synopses," vol. 11, no. 9, pp. 1016–1028. [Online]. Available: https://dl.acm.org/doi/10.14778/3213880.3213882

[28] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita, "Improved histograms for selectivity estimation of range predicates," in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data - SIGMOD '96*. ACM Press, pp. 294–305.

[29] M. Muralikrishna and D. J. DeWitt, "Equi-depth multidimensional histograms," vol. 17, no. 3, pp. 28–36. [Online]. Available: https://dl.acm.org/doi/10.1145/971701.50205

[30] Z. Fong, "The design and implementation of the postgres query optimizer."

[31] P. G. D. Group, "PostgreSQL 17.2 Documentation." [Online]. Available: https://www.postgresql.org/files/documentation/pdf/17/postgresql-17-A4.pdf

[32] V. Leis, B. Radke, A. Gubichev, A. Kemper, and T. Neumann, "Cardinality Estimation Done Right: Index-Based Join Sampling."

[33] P. J. Haas, J. F. Naughton, S. Seshadri, and A. N. Swami, "Selectivity and Cost Estimation for Joins Based on Random Sampling," vol. 52, no. 3, pp. 550–569. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0022000096900410

[34] R. J. Lipton and J. F. Naughton, "Query size estimation by adaptive sampling," in *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, ser. PODS '90. Association for Computing

Machinery, pp. 40–46. [Online]. Available: https://dl.acm.org/doi/10.1145/298514.298540

[35] R. J. Lipton, J. F. Naughton, and D. A. Schneider, "Practical selectivity estimation through adaptive sampling," in *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, ser. Sigmod '90. Association for Computing Machinery, pp. 1–11. [Online]. Available: https://doi.org/10.1145/93597.93611

[36] M. Müller, "Selected Problems in Cardinality Estimation." [Online]. Available: https://madoc.bib.uni-mannheim.de/63683/1/Dissertation_Magnus_Mueller.pdf

[37] A. Kipf, T. Kipf, B. Radke, V. Leis, P. Boncz, and A. Kemper. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. [Online]. Available: http://arxiv.org/abs/1809.00677

[38] S. Cluet and G. Moerkotte, "On the complexity of generating optimal left-deep processing trees with cross products," in *Database Theory — ICDT '95*, G. Gottlob and M. Y. Vardi, Eds. Springer Berlin Heidelberg, vol. 893, pp. 54–67. [Online]. Available: http://link.springer.com/10.1007/3-540-58907-4_6

[39] G. Moerkotte, T. Neumann, and G. Steidl, "Preventing bad plans by bounding the impact of cardinality estimation errors," vol. 2, no. 1, pp. 982–993. [Online]. Available: https://dl.acm.org/doi/10.14778/1687627.1687738

[40] R. Bellman, *Dynamic Programming*. Princeton Univ. Pr.

[41] B. Radke and T. Neumann, "LinDP++: Generalizing Linearized DP to Crossproducts and Non-Inner Joins."

[42] G. Moerkotte and T. 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*, pp. 930–941.

[43] G. Moerkotte, "DP-Counter Analytics," p. 61. [Online]. Available: https://madoc.bib.uni-mannheim.de/1156/1/MA_TR_2006_002.pdf

[44] G. Moerkotte and T. Neumann, "Dynamic programming strikes back," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. ACM, pp. 539–552. [Online]. Available: https://dl.acm.org/doi/10.1145/1376616.1376672

[45] A. Meister, "Parallel Join Ordering." [Online]. Available: https://www.dbse.ovgu.de/-p-578/_/2_join_ordering.pdf

[46] B. Vance and D. Maier, "Rapid bushy join-order optimization with Cartesian products," vol. 25, no. 2, pp. 35–46. [Online]. Available: https://dl.acm.org/doi/10.1145/235968.233317

[47] B. Vance, "Join-order Optimization with Cartesian Products."

[48] T. Ibaraki and T. Kameda, "On the optimal nesting order for computing N-relational joins," vol. 9, no. 3, pp. 482–502. [Online]. Available: https://dl.acm.org/doi/10.1145/1270.1498

[49] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT press.

[50] T. Neumann and B. Radke, "Adaptive Optimization of Very Large Join Queries," in *Proceedings of the 2018 International Conference on Management of Data*. ACM, pp. 677–692. [Online]. Available: https://dl.acm.org/doi/10.1145/3183713.3183733