On the Performance of Query Optimization Without Cost Functions and Very Simple Cardinality Estimation

Daniel Flachs
University of Mannheim
Mannheim, Germany
email: flachs@uni-mannheim.de

Guido Moerkotte
University of Mannheim
Mannheim, Germany
email: moerkotte@uni-mannheim.de

***Abstract*—In order to enable a fast time to market for new Database Management Systems (DBMS), we introduce two simple, very easy to implement cardinality estimators and a build-plan method that does not require any cost function. Experimentally, we demonstrate that different plan generators incorporating these ideas are quite competitive on the Join Order Benchmark (JOB): the join ordering algorithm DPccp yields plans that are at most a factor of 2.10 away from the optimum without using any runtime-dependent cost function if cardinalities are known. Thus, using our approach obviates the effort of implementing sophisticated cardinality estimation methods and cost functions in a first version of a DBMS.**

Keywords-query optimization; hash join; cardinality estimation; cost function; plan generation.

I. Introduction

Developing a new DBMS like DuckDB [1] from scratch is a challenging task. To achieve a short time to market, compromises in different modules of the system are required. Two modules that are tedious to implement and test in the context of query optimization are cardinality estimation (CE) and cost functions (CF).

We observed recent attempts in the literature to simplify query optimization [2][3]. However, these approaches are often monolithic in the sense that they touch many of the different 'moving parts' involved in query optimization, without proper modularization. Those modules are

- the **join ordering algorithm** that decides the logical join order,
- the **build-plan procedure** (BP) used for selecting the most suitable physical (join) operator,
- the **cardinality estimator** (CE) that provides a cardinality estimate for any subset of relations considered by the plan generator,
- the **cost function** (CF) to compare the cost of two plan alternatives.

If implemented correctly, the modules are independent of each other and can be exchanged without changing the rest of the system, allowing for an extensive evaluation of the different combinations.

In this paper, we pose the questions: How simple can cardinality estimation, cost function, and build plan procedure become while still yielding query evaluation plans (QEPs) with an acceptable quality? Can we get acceptable plans even if BP does not use any cost function at all?

In order to answer these questions, we propose two very simple cardinality estimators: CE_{base} and CE_{sel}. Further, we propose a new build-plan method for query optimization (BP_{smart}) that does not require any cost function and instead relies only on cardinality estimates for its decisions. This is in stark contrast to the traditional build-plan procedure BP_{trad}: Given two (sub-) plans to be joined, it decides on the argument order (e.g., build vs. probe side for hash joins) as well as on the actual join implementation to be used in a purely cost-based manner.

In order to evaluate and compare the performance of the newly proposed cardinality estimators and build-plan method, we implemented the Plan Generator Benchmarking Framework *PgBench*. It allows to orthogonally test the performance of combinations of join ordering algorithms, cardinality estimation methods, cost functions, and build-plan procedures. In this paper's evaluation, we report on the performance of the join ordering algorithms DPccp [4], GooCost [5], and GooCard [6][7]. As cardinality estimation methods, we use our new cardinality estimators CE_{base} and CE_{sel} as well as the existing $CE_{IA\text{-}M}$, which relies on the independence assumption, and CE_{tru}, which provides the true cardinalities. As cost functions, we use CF_{tru} and CF_{est} for the true and estimated execution costs. The build-plan procedures used are BP_{trad} and the newly designed BP_{smart}. As the set of queries, we use the Join Order Benchmark (JOB) [8].

Since the implementations of BP_{smart} and CE_{base} or CE_{sel} require only little effort while being quite competitive, as seen in the evaluation, this will help to achieve a short time to market for a new DBMS.

The rest of the paper is organized as follows. Section II presents basic notions like plan class, ccp, uniqueness, and loss factor of a plan. Section III gives the details on the new cardinality estimation methods CE_{base} and CE_{sel} as well as for $CE_{IA\text{-}M}$. Section IV introduces the different join implementations as well as the derivation of the cost functions CF_{tru} and CF_{est} for them. Sections V and VI introduce the two build-plan procedures BP_{trad} and BP_{smart}. Section VII contains the evaluation. An overview of related work is given in Section VIII. Section IX concludes the paper.

II. Preliminaries

For a given conjunctive query, we denote by $\mathcal{R} := \{R_1, \ldots, R_n\}$ the set of relations in its `from`-clause. The set of

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

attributes of R_i is denoted by $\mathcal{A}(R_i)$. The single-table selection predicates for R_i are denoted by p_i. Equijoin predicates for a join between R_i and R_j are denoted by $p_{i,j}$. The attributes accessed by a selection or join predicate p are denoted by $\mathcal{F}(p)$. The *query graph (QG)* has the relations in $\mathcal{R}$ as nodes and an edge connecting R_i and R_j for every join predicate $p_{i,j}$. A *plan class* $S \subseteq \mathcal{R}$ is a subset of relations inducing a connected subgraph of the query graph. In this case, we write $\text{pc}(S)$. The set of attributes of a plan class $\text{pc}(S)$ is defined as $\mathcal{A}(S) := \bigcup_{R_i \in S} \mathcal{A}(R_i)$. Two subsets $S_1, S_2 \subseteq \mathcal{R}$ form a *ccp* if $\text{pc}(S_1)$, $\text{pc}(S_2)$, $S_1 \cap S_2 = \emptyset$, and there is at least one join predicate $p_{i,j}$ such that $R_i \in S_1$ and $R_j \in S_2$ [4]. In this case, we write $\text{ccp}(S_1, S_2)$. The abbreviation ccp stands for a *csg-cmp-pair*, which refers to a pair of connected subgraphs *(csg)* of the query graph that are complements *(cmp)* to each other. The notion of a ccp was first introduced by Moerkotte and Neumann [4]. To find the optimal join order using dynamic programming, all such pairs must be enumerated.

For both our new build-plan procedure and our new cardinality estimators, we need to determine whether the join predicate connecting the two sets of a $\text{ccp}(S_1, S_2)$ implies uniqueness on S_1 and/or S_2.

The join predicate $P(S_1, S_2)$ for $\text{ccp}(S_1, S_2)$ is the conjunction of all $p_{i,j}$ such that $R_i \in S_1$ and $R_j \in S_2$. The set of join attributes of S_i ($i = 1, 2$) is then defined as $\mathtt{J}(S_i, (S_1, S_2)) := \mathcal{F}(P(S_1, S_2)) \cap \mathcal{A}(S_i)$. We say that $\text{ccp}(S_1, S_2)$ determines S_i uniquely if $\mathtt{J}(S_i, (S_1, S_2))$ is a (super-) key of S_i. If $\mathtt{K}(S_i)$ denotes the set of keys of S_i, we can express this as

$$\mathfrak{U}(S_i, (S_1, S_2)) := \exists \kappa \in \mathtt{K}(S_i) : \kappa \subseteq \mathtt{J}(S_i, (S_1, S_2)).$$

In order to derive this uniqueness, we need to determine the keys for plan classes S. We start by assuming that for every relation R_i the set of primary and secondary keys is given as $\mathtt{K}(\{R_i\})$. For a $\text{ccp}(S_1, S_2)$, we can determine the set of keys $\mathtt{K}(S_1 \cup S_2)$ as follows. If there exists a key $\kappa \in \mathtt{K}(S_i)$ such that $\kappa \subseteq \mathtt{J}(S_i, (S_1, S_2))$, then $\mathtt{K}(S_{3-i}) \subseteq \mathtt{K}(S_1 \cup S_2)$. Note that S_{3-i} refers to S_2 for $i = 1$ and vice versa. If this is true for neither S_1 nor S_2, then $\mathtt{K}(S_1 \cup S_2)$ contains $\kappa_1 \cup \kappa_2$ for all $\kappa_1 \in \mathtt{K}(S_1)$ and for all $\kappa_2 \in \mathtt{K}(S_2)$.

To measure the error between a true value and an estimate, we use the *q-error* [9]. Let $x > 0$ be some true value and $\hat{x} > 0$ be its estimate, then $\text{qerr}(x, \hat{x}) := \max\left(\frac{x}{\hat{x}}, \frac{\hat{x}}{x}\right)$.

Finally, we define the *loss factor* of a plan. Given a plan class S and a plan P for S, we define the *loss factor* of P as the true cost of P divided by the true cost of the overall best plan for S. This is a value greater or equal to 1.

III. Cardinality Estimation

In this section, we present two new cardinality estimators (CE_{base} and CE_{sel}) which only differ in their treatment of single-table selection predicates. CE_{base} ignores them, whereas CE_{sel} takes them into account. We start by defining both estimators for plan classes containing a single relation $R_i \in \mathcal{R}$:

$$\text{CE}_{\text{X}}(\{R_i\}) := \begin{cases} |R_i| & \text{if X = 'base'} \\ |\sigma_{p_i}(R_i)| & \text{if X = 'sel'} \end{cases} \quad (1)$$

Then, for general plan classes, we define

$$\text{CE}_{\text{X}}(S) := \min_{\text{ccp}(S_1,S_2):\ S=S_1 \cup S_2} \text{CE}_{\text{X}}(S, (S_1, S_2)) \quad (2)$$

where, with $\mathfrak{U}(S_i)$ abbreviating $\mathfrak{U}(S_i, (S_1, S_2))$,

$$\text{CE}_{\text{X}}(S, (S_1, S_2)) := \begin{cases} \min(\text{CE}_{\text{X}}(S_1), \text{CE}_{\text{X}}(S_2)) & \text{if } \mathfrak{U}(S_1) \wedge \mathfrak{U}(S_2) \\ \text{CE}_{\text{X}}(S_1) & \text{if } \neg\mathfrak{U}(S_1) \wedge \mathfrak{U}(S_2) \\ \text{CE}_{\text{X}}(S_2) & \text{if } \mathfrak{U}(S_1) \wedge \neg\mathfrak{U}(S_2) \\ \text{CE}_{\text{X}}(S_1) \cdot \text{CE}_{\text{X}}(S_2) & \text{if } \neg\mathfrak{U}(S_1) \wedge \neg\mathfrak{U}(S_2) \end{cases}$$

The idea is to use the smaller cardinality if both arguments are unique, the cardinality of the cross product if neither argument is unique, and the cardinality of the non-unique side if one of the argument relations is unique but not the other. Clearly, both cardinality estimators may produce overestimates and never produce underestimates, given true inputs. Note that for an implementation of CE_{sel}, an estimation procedure for $|\sigma_{p_i}(R_i)|$ is required. We propose to use sampling, as it is easy to implement and universally applicable [10]–[12].

For comparison in our evaluation, we also use the cardinality estimator $\text{CE}_{\text{IA-M}}$, which applies the independence assumption using the multiplicative rule [13]:

$$\text{CE}_{\text{IA-M}}(S) := \prod_{R_i \in S} |\sigma_{p_i}(R_i)| \cdot \prod_{p_{i,j}:\ R_i, R_j \in S} \text{sel}(p_{i,j}) \quad (3)$$

where $\text{sel}(p_{i,j}) := \frac{|R_i \bowtie_{p_{i,j}} R_j|}{|R_i| \cdot |R_j|}$ is the (true) selectivity of $p_{i,j}$.

IV. Cost Functions

This section first introduces the two physical hash join operators and their variants before outlining how their cost functions are derived from runtime experiments.

A. The Join Implementations

We consider two physical main-memory hash join operators: the *chaining hash join (CH-join)* and the *3D hash join (3D-join)* [14]. Their main difference is the hash table data structure which is built and probed during the join. The CH-join uses a hash table that resolves collisions by collecting all colliding keys into one linked list for each hash table bucket. The 3D-join uses a 3D hash table that groups duplicate keys together in a hierarchical collision chain organization with main and sub nodes. Further, for both physical operators, we consider two variants for the physical design of collision chain nodes, and three prefetching variants.

For the **collision chain node design**, there is an *unpacked (upk)* and a *packed (pkd)* variant. The unpacked variant is the original implementation [14], where each (main, sub) collision chain node stores one tuple pointer. The idea behind the packed variant is to improve the cache line utilization of a single collision chain node. Here, each collision chain node of the CH-join can store three tuple pointers. For the 3D-join, each main node stores five tuple pointers with equal join attribute values, and each sub node stores three tuple pointers.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Prefetching is a known technique to hide memory latencies of cache misses [15][16]. We therefore augment the non-prefetching implementations *(NoPF)* of the physical join operators by two prefetching approaches: *rolling prefetching (RoPF)* and *asynchronous memory access chaining (AMAC)* [16]. AMAC maintains a small ring buffer that keeps track of the processing state of tuples during the build or probe phase of a hash join, where the number of states depends on the join phase and physical hash table implementation. Before the next processing step of buffer element i, e.g., accessing a hash table bucket or inserting into a hash table collision chain node, a prefetch is issued for the necessary memory address, and processing continues with the next buffer element. Only after all other ring buffer elements have been examined, processing continues for buffer element i, giving the prefetch issued by i time to be completed. In AMAC, both hash table directory entries and hash table collision chain nodes are prefetched. There is obviously a tradeoff between the time saved due to hidden latencies, and the time lost due to branch misprediction penalties for handling the different AMAC states. As a compromise between NoPF and AMAC, we also implemented RoPF, which only prefetches the hash table directory entries, but not the collision chain nodes, simplifying the prefetching logic.

In summary, for each join in the plan, we can choose between any of the following 36 physical operators and implementations:

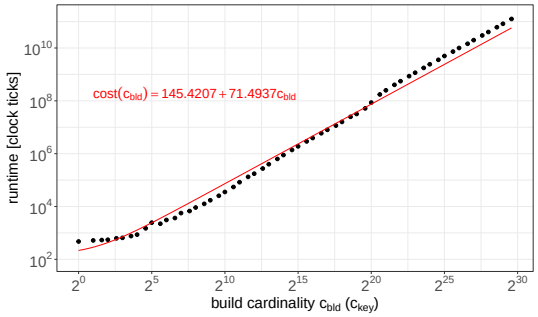
$$\{\text{CH-join}, \text{3D-join}\} \times \{\text{upk}, \text{pkd}\} \times \{\text{NoPF}, \text{RoPF}, \text{AMAC}\}^2$$

Note that the prefetching variants can be applied independently to the build and probe phase of a single hash join (hence the squared term), while both phases must agree on the physical node design.

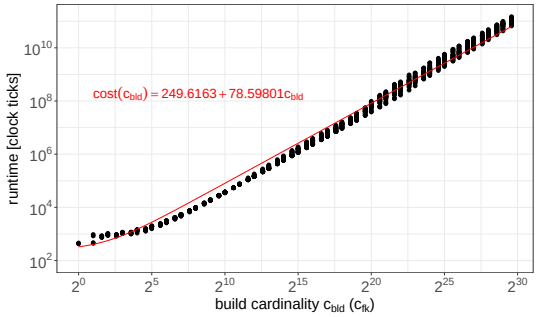
B. Derivation of Cost Functions

The cost functions in this paper are constructed from measurements of runtime experiments. We therefore first briefly outline the experimental setup to obtain the measurements before describing the process of constructing cost functions from them.

1) Runtime Experiments: We measure the runtime of a single key/foreign key join between a key relation R and a foreign key relation S separately for the build and probe phase. The cardinalities for R and S are varied between 2^0 and 2^{30} in half-steps of powers of two, i.e., $\{t \cdot 2^i \mid i \in [0, 30], t \in \{1, 1.5\}\}$. For validation purposes, we additionally measured runtimes for $t \in \{1.3, 1.7\}$ that were not used for cost function generation. To generate the foreign keys for S, we draw $|S|$ random samples according to a (1) uniform and (2) standard Zipf distribution from a domain $[0, |S|/2^d - 1]$ with $d \in [0, 10]$ for build, and $d \in [0, 6]$ for probe. This results in a total of 43 489 input parameter combinations (without validation). For each of these combinations, we evaluate $|\{\text{CH}, \text{3D}\} \times \{\text{upk}, \text{pkd}\} \times \{\text{NoPF}, \text{RoPF}, \text{AMAC}\}| = 12$ build and the same number of probe runs. The runtimes are recorded in terms of timestamp counter clock ticks, a proxy for the wall-clock time [17, Chap. 18.17].



(a) unique build.



(b) non-unique build.

Figure 1. Measured runtimes and approximated functional cost functions for the build phase of CH-upk-NoPF.

2) Cost Function Generation: To turn the discrete data points from our runtime experiments into continuous cost functions usable by the query optimizer, we apply existing methods [9][18] to find an approximation function from a set of functions (e.g., constants, linear functions, or polynomials of a certain degree) that minimizes the maximum q-error between the true measured value and the values given by the function.

For each of the physical operator implementation variants from Section IV-A, we create separate cost functions for each case from $\{\text{build}, \text{probe}\} \times \{\text{unique}, \text{non-unique}\}$, i.e., for the two join phases and depending on the uniqueness of the build side. Each of the cost functions uses a subset of the following input parameters: the join's input build, probe and output cardinality (c_{bld}, c_{prb}, c_{res}), and the number of distinct values of the foreign key attribute ($nodv$).

Further, we construct cost functions in two levels of precision: a more precise, but also more complex *tabulated* cost function, and a less precise, but less complex *functional* cost function.

a) Functional Cost Functions: The functional cost functions capture all measurements of a single case (build/probe, uniqueness) in a single mathematical function. All cost functions are linear. For the build phase, the cost functions are 1-dimensional with c_{bld} as input, while they are 3-dimensional for the probe phase and use all three join cardinalities, c_{bld}, c_{prb}, and c_{res}. Figure 1 shows the measured runtimes from our experiments for the CH-join (unpacked, no prefetching) alongside the respective cost functions generated by approximation. Observe that the cost functions produce both over- and underestimates.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

b) Tabulated Cost Functions: The tabulated cost functions use one- or two-dimensional lookup tables (hence the name) to map a subset of the measurements to an n-dimensional approximation function for that subset. To compute cost values for input parameter values between lookup table entries, we apply (bi-) linear interpolation. If the requested values are above the maximum values in the table, linear extrapolation is applied.

First, consider the build phase. For unique builds, each lookup table entry simply maps c_{bld} to the respective experimental runtime (a constant). For non-unique builds, each c_{bld} is associated with either a constant or a linear function that takes $nodv$ as its only input.

All lookup tables for the probe phase are two-dimensional in c_{bld} and c_{prb}. In the case of unique build sides, each lookup table entry contains a constant or a linear function that takes $nodv$ as its only input. For non-unique builds, each $(c_{\text{bld}}, c_{\text{prb}})$-pair is associated with either a constant or a two-dimensional linear function in $nodv$ and c_{res}.

We use the more precise tabulated cost functions as CF_{tru} and the simpler but less precise functional cost functions as CF_{est}. Note that both CF_{tru} and CF_{est} exhibit errors with regard to the true execution cost of the respective join, even though the name CF_{tru} might suggest otherwise. We merely assume CF_{tru} to be the true execution cost in order to have a notion of 'true optimality'. This distinction, however, is of minor importance towards the evaluation in Section VII, as the maximum q-error of CF_{tru} is well below 2 across all runtime measurements.

V. Traditional Build Plan

The task of finding a join tree for a given query graph can be split into two distinct problems: the *join ordering* that decides which relations and subtrees to join next, and the *operator selection* that chooses the most suitable physical operator and argument order to join two subtrees. The former problem is tackled by optimal join order ordering algorithms like DPccp [4], or heuristics, like GooCard [6][7] and GooCost [5].

Algorithm 1 DPccp [4].

```
1:  function DPccp
2:    Input: a connected QG w/ relations R = {R_1, ..., R_n}
3:    Output: an optimal bushy join tree
4:    for all R_i ∈ R do BestPlan({R_i}) ← R_i
5:    for all ccp(S_1, S_2), S ← S_1 ∪ S_2 do
6:      T_1 ← BestPlan(S_1), T_2 ← BestPlan(S_2)
7:      T_curr ← BuildPlan(T_1, T_2)
8:      if cost(BestPlan(S)) > cost(T_curr) then
9:        BestPlan(S) ← T_curr
10:   return BestPlan(R)
```

For illustration, we show DPccp in Algorithm 1: it iterates over each ccp and stores the (cost-wise) optimal join tree for each plan class in a data structure. The operator selection problem is decided by a build-plan subroutine (called in Line 7), like BP_{trad}, shown in Algorithm 2, or BP_{smart} (see Section VI). Build-plan decides which physical join operators (*Implementations* in Algorithm 2, Line 5) are considered, and which argument order is better, $T_1 \bowtie T_2$ or $T_2 \bowtie T_1$. In case of BP_{trad}, this requires a total of 72 cost function evaluations for both join argument orders and 36 physical operator variants (see Section IV-A).

Algorithm 2 BP_{trad} [19, p. 62].

```
1:  function BuildPlanTrad(T_1, T_2)
2:    Input: two join trees T_1, T_2
3:    Output: the best join tree for joining T_1 and T_2
4:    BestTree ← null, cost(BestTree) ← ∞
5:    for each impl ∈ Implementations do
6:      T ← T_1 ⋈^impl T_2
7:      if cost(BestTree) > cost(T) then
8:        BestTree ← T
9:      T ← T_2 ⋈^impl T_1
10:     if cost(BestTree) > cost(T) then
11:       BestTree ← T
12:   return BestTree
```

TABLE I. Maximum loss factors of two plans for different input cardinality situations.

	CH-upk-(RoPF, RoPF)-bun	3D-upk-(RoPF, AMAC)-bnu
$2\|R\| < \|S\|$	1.79	7.71
$2\|R\| = \|S\|$	1.54	2.63
$\|R\| = \|S\|$	1.40	2.84
$\|R\| = 2\|S\|$	1.43	2.55
$\|R\| > 2\|S\|$	4.45	2.01

Both join ordering and build-plan apply cost functions to decide on the best alternative. Note that both could use different cost functions independently of each other: Build-plan could use cost functions based on join runtime (Section IV-B), whereas DPccp could pick the partial plan based on CF_{cout} (minimizing the sum of the cardinalities of the intermediate results) [20]. This flexibility is exploited for BP_{smart} in our evaluation.

VI. Smart Build Plan

It is the main goal of the build-plan procedure BP_{smart} to make all required decisions based only on the cardinalities of the input relations to the join, i.e., without any reference to a cost function. At the same time, it should try to minimize the loss factor of the produced (partial) plan.

One common heuristics is to always build on the smaller input relation. However, we will see that we need to slightly relax this rule when determining the order of the join's arguments.

Having chosen the build relation, we need to decide which join implementation and variant to apply. Based on runtime experiments [14], we come up with the following rule: If the join attributes of the build side cover a key of the build side (i.e., unique builds, *bun* for short), the CH-join is the clear favorite. Otherwise, for non-unique builds (*bnu* for short), the 3D-join is used. Since the experiments indicate that the packed versions of the algorithms only provide limited improvements

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

in rare cases (many duplicates, uniform distribution), BP_{smart} uses only the unpacked versions.

For the CH-join with unique build sides, rolling prefetching is superior to AMAC in most cases. Further, it provides only little overhead for small builds compared to no prefetching. This applies to both the build and the probe phase. For the 3D-join with non-unique build-sides, we found that rolling prefetching is the best compromise for build and AMAC the best compromise for probe.

Let us come back to the problem of deciding which relation to use for the build. If we have a key relation R and a foreign key relation S, we can use the CH-join with build on R and the 3D-join with build on S. For a key/foreign key join, Table I contains the maximum loss factor of these two plans for different cardinality situations of the input relations. The header line shows the two physical plans used. For instance, 3D-upk-(RoPF, AMAC)-bnu refers to the 3D-join in the unpacked variant with a non-unique build side using rolling prefetching for build and AMAC for probe. The table indicates that we should use the CH-join if the cardinality of the key relation does not exceed twice the cardinality of the foreign key relation. If both or none of the input relations are unique, BP_{smart} uses the smaller one as the build relation.

The details of BP_{smart} are given in Algorithm 3. By convention, the right-hand side of the join symbol is the build side. The function *relset* returns the set of relations joined in a (partial) plan. Further, we abbreviate CH-upk-(RoPF, RoPF) by ch and 3D-upk-(RoPF, AMAC) by $3d$. We always use a CH-join if the build is unique, otherwise we use the 3D-join. If neither or both of the input relations are unique, then the build is on the smaller input. In Line 16, we make sure that T_1 is unique and T_2 is non-unique by a conditional swap. Then, we proceed as deduced in the above analysis.

Algorithm 3 BP_{smart}.

```
1:  function BUILDPLANSMART(T1, T2)
2:      Input: two join trees T1, T2
3:      Output: a join tree for joining T1 and T2
4:      if 𝔘(relset(T1)) ∧ 𝔘(relset(T2)) then
5:          if card(T1) ≤ card(T2) then
6:              ResultTree ← T2 ⋈^ch T1
7:          else
8:              ResultTree ← T1 ⋈^ch T2
9:          return ResultTree
10:     if ¬𝔘(relset(T1)) ∧ ¬𝔘(relset(T2)) then
11:         if card(T1) ≤ card(T2) then
12:             ResultTree ← T2 ⋈^3d T1
13:         else
14:             ResultTree ← T1 ⋈^3d T2
15:         return ResultTree
16:     if 𝔘(relset(T2)) then swap(T1, T2)
17:     if card(T1) ≤ 2 · card(T2) then
18:         ResultTree ← T2 ⋈^ch T1
19:     else
20:         ResultTree ← T1 ⋈^3d T2
21:     return ResultTree
```

VII. Evaluation

As our evaluation dataset and workload, we use the Join Order Benchmark (JOB) [8][21]. It consists of 33 query templates on the Internet Movie Database (IMDb) schema and dataset, which are instantiated as 113 queries, each with four to 17 relations. Queries from the same template only differ in their conjunctively connected single-table filter predicates. JOB's challenging analytical select-project-join queries justify its frequent use in the literature to evaluate the quality of plan generators [2][3][22]–[24].

In order to answer the question from the introduction whether the generation of acceptable plans is possible without a runtime-related cost function, we consider two cases in particular. Recall that our BP_{smart} makes all decisions without any notion of cost. We consider the combination of BP_{smart} with DPccp and CF_{cout}, and with GooCard, entirely eliminating runtime-related cost functions from the process of plan generation. We compare this to the usual approach where both join ordering and build-plan rely on runtime-related cost functions like CF_{tru} and CF_{est}.

Before going into a more detailed analysis, let us illustrate the general impact of join ordering on the given workload: If we modify plan generation such that it produces the overall

TABLE II. Loss Factors for DPccp, GooCard, and GooCost.

DPccp					
BP	CF	CE_{tru}	$CE_{IA\text{-}M}$	CE_{base}	CE_{sel}
trad	tru	1.00	10.51	16.47	7.39
		1.00	1.39	2.32	1.98
	est	3.99	3.02	6.90	6.18
		1.82	1.51	2.67	2.57
smart	tru	2.52	17.88	6.84	6.09
		1.43	1.75	2.45	2.38
	est	2.10	20.98	6.94	6.09
		1.38	1.65	2.41	2.30
smart	cout	2.10	18.11	6.10	6.10
		1.41	2.07	2.57	2.31
GooCost					
BP	CF	CE_{tru}	$CE_{IA\text{-}M}$	CE_{base}	CE_{sel}
trad	tru	2.18	2.51	7.39	8.80
		1.11	1.32	2.10	2.19
	est	4.54	3.66	8.92	8.20
		1.97	1.72	2.57	2.52
smart	tru	2.86	2.32	6.90	12.23
		1.57	1.57	2.17	2.39
	est	2.66	2.66	6.90	5.74
		1.45	1.49	2.14	2.17
GooCard					
BP	CF	CE_{tru}	$CE_{IA\text{-}M}$	CE_{base}	CE_{sel}
trad	tru	1.43	13.52	7.39	13.65
		1.06	1.51	2.16	2.35
	est	3.15	6.52	8.92	8.30
		1.73	1.78	2.59	2.61
smart	—	2.10	15.53	6.90	6.71
		1.42	1.94	2.27	2.32

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

worst possible join order (using DPccp with cost maximization), the maximum (average) loss factor across all JOB queries is 35 785 (1168). As we will see, all subsequent loss factors are orders of magnitude away from this worst case.

Table II contains the plan loss factors for DPccp, GooCost, and GooCard. The first two columns indicate which combination of build-plan (BP) and cost function (CF) was applied. Typically, BP_{trad} uses the same cost function as the join ordering algorithm. In contrast, BP_{smart} does not require a cost function, so CF refers *only* to the join ordering algorithm for these cases. For every BP-CF-combination, there exist two lines. The first (second) line contains the maximum (average) loss factor taken over all JOB queries. The four numbers in each row correspond to the cardinality estimator used, as shown in the header line.

To start our discussion, we consider the first two lines of Table II. Here, we evaluate the loss factor of DPccp under BP_{trad} and CF_{tru} for different cardinality estimators. We see that the maximum loss factor under CE_{tru} is 1, which is the smallest possible value. The maximum (average) loss factors for the different cardinality estimators are 10.51 (1.39) for $CE_{IA\text{-}M}$, 16.47 (2.32) for CE_{base}, and 7.39 (1.98) for CE_{sel}. Thus, CE_{base} is the worst for the maximum loss factor and CE_{sel} the best. For the average, $CE_{IA\text{-}M}$ is the best. However, we still use CF_{tru} here, thus no cost function errors occur.

The next two lines are for BP_{trad} and CF_{est}. Here, the more realistic case of an erroneous cost function is evaluated. We see that even for the true cardinalities CE_{tru}, the maximum loss factor is 3.99 and the average is 1.82. Interestingly, the errors of $CE_{IA\text{-}M}$ and CF_{est} seem to compensate each other in the worst case, since the maximum loss factor decreases to 3.02, whereas the average increases to 1.51. For our new cardinality estimators, we have maximum loss factors of 6.90 (CE_{base}) and 6.18 (CE_{sel}), and average loss factors of 2.67 (CE_{base}) and 2.57 (CE_{sel}). Thus, they perform worse than $CE_{IA\text{-}M}$ if BP_{trad} and CF_{est} are in use.

This picture changes if we consider our newly introduced build-plan procedure BP_{smart}, where DPccp uses CF_{est}. Here, the maximum loss factor of $CE_{IA\text{-}M}$ (20.98) is far worse than that of CE_{base} (6.94) and CE_{sel} (6.09). On average, however, $CE_{IA\text{-}M}$ performs slightly better than these.

One of the goals of this paper is to provide a possibility to generate plans without the need for any runtime-related cost functions as, e.g., constructed in Section IV. For that purpose, we evaluated DPccp using CF_{cout} [20], which sums up the intermediate result sizes of joins as provided by the cardinality estimator in place. Further, BP_{smart} makes all decisions based only on cardinalities and uniqueness properties. Thus, we next report on the performance of DPccp without any reference to a runtime-related cost function. The last two lines of the DPccp-block contain the loss factors for this scenario. We see that if true cardinalities are used, the maximum loss factor is only 2.10 with an average of 1.41. If $CE_{IA\text{-}M}$ is used, the maximum loss factor increases to 18.11, which is much higher than the worst case for CE_{base} and CE_{sel}. On average, $CE_{IA\text{-}M}$ performs slightly better than CE_{base} and CE_{sel}. Further, under these conditions, both CE_{base} and CE_{sel} perform slightly better than under BP_{trad} and CF_{est}. Comparing CE_{base} and CE_{sel}, we see that the maximum loss factor is the same, but CE_{sel} performs slightly better than CE_{base} on average. This indicates that in a first version of a new DBMS, one could use CE_{base}. In some later version, CE_{sel} could be implemented. Remember that CE_{sel} requires the estimation of single-table selection predicates, for which, e.g., sampling needs to be implemented.

Let us now turn to the heuristics GooCost and GooCard. Under optimal conditions (BP_{trad}, CF_{tru}, CE_{tru}), the maximum (average) loss factors of GooCost is 2.18 (1.11) and for GooCard 1.43 (1.06). Thus, we can conclude that under optimal conditions, GooCard outperforms GooCost. For heuristics, both perform quite well. Using the estimated costs CF_{est} in BP_{trad} instead, these numbers increase to 3.15 (1.73) for GooCard with CE_{tru}. For BP_{smart}, the loss drops to 2.10 (1.32). It might not be intuitive why GooCard produces different loss factors for CF_{tru} and CF_{est}, although it only uses cardinalities and not costs for its join ordering decisions. In this particular case, the cost function is used by BP_{trad}. This also explains why there is no cost function shown for GooCard and BP_{smart}, as neither needs a notion of cost.

Turning to erroneous cardinality estimators for GooCard using BP_{smart}, we see that $CE_{IA\text{-}M}$ performs worse (15.53/1.94) than both CE_{base} (6.90/2.27) and CE_{sel} (6.71/2.32). No runtime-related cost function is needed here, similar to DPccp with BP_{smart} (last two lines of the DPccp-block). Comparing these, we see that going from DPccp to GooCard only slightly increases the maximum loss factor, while the average loss factor decreases for CE_{base} and remains about the same for CE_{sel}. CE_{base} has a slightly higher worst case than CE_{sel}.

Most DBMSs provide at least two different join ordering algorithms: one like DPccp for queries with moderate numbers of relations (say at most 15–20), and one heuristics for larger queries. The above numbers suggest that we can obviate DPccp and only implement GooCard in a first version of a newly developed DBMS without compromising performance too much, if we use CE_{base} or CE_{sel}. If we compare this scenario to the one where we implemented the cardinality estimator $CE_{IA\text{-}M}$ and some cost function CF_{est}, as well as the join ordering algorithm DPccp with the build-plan procedure BP_{trad}, we see that the combination of GooCard and BP_{smart} loses only a factor of about 2 in the worst case (going from 3.02 to 6.90/6.71 for CE_{base}/CE_{sel}) and a factor of $1.70 = 2.57/1.51$ for CE_{base} and $1.50 = 2.31/1.51$ for CE_{sel} on the average case.

VIII. Related Work

Simplification of query optimizers is not a new idea. For example, Datta et al. propose the algorithm Simpli-Squared for join ordering without cardinality and cost estimation [2]. The basic idea is to first execute key/foreign key joins and then n:m-joins. Since no notion of cost/cardinality is available to Simpli-Squared, the ordering of relations for a star-query is random, depending on the order in which key/foreign key joins are enumerated (e.g., depending on the order of the relations in the `from`-clause).

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Another example for query optimizer simplification is proposed by Hertzschuch et al. [3][25]. However, their approach highly intertwines the proposed cardinality estimation procedure with a newly proposed join ordering heuristics. Further, besides some cardinality estimates for filtered base relations, it also requires knowledge about the maximum multiplicity of the distinct values in the join attributes. Thus, it is much more complex than our cardinality estimators which are, in contrast, also independent of the join enumeration algorithm.

Notably, neither approach uses proper cost functions and leaves significant parts of the plan generation to PostgreSQL, relying on PostgreSQL's simple cost model whose errors remain unknown.

IX. Conclusion and Future Work

Since implementing and testing cost functions can be quite tedious, we showed that we can implement a competitive plan generator that does not rely on any cost function. Instead, the new build-plan procedure BP_{smart} makes all decisions based only on cardinality estimates. Further, we demonstrated that a very simple cardinality estimator CE_{base}, which does not even require cardinality estimation for single-table selection predicates, is quite competitive. Our evaluation shows that if the effort is undertaken to implement cardinality estimation for this case, e.g., based on sampling, then the average loss factor decreases when using CE_{sel}. Last but not least, we showed that implementing only GooCard and no other join ordering algorithm with optimality guarantees like DPccp results in only a limited loss of plan quality. Taking these findings together allows for an easy to implement query optimizer enabling a short time to market for a new DBMS.

For future work, we intend to perform an in-depth analysis on the granularity of individual queries and plans to extend our evaluation of plan quality.

Acknowledgments

We would like to thank Simone Kehrberg for proofreading the paper, and Nazanin Rashedi for helpful comments and discussions. We would also like to thank the anonymous reviewers for their suggestions to improve the paper.

References

[1] M. Raasveldt and H. Mühleisen, "DuckDB: An embeddable analytical database", in *Proc. of the ACM SIGMOD Conf. on Management of Data*, 2019, pp. 1981–1984.

[2] A. Datta, B. Tsan, Y. Izenov, and F. Rusu, "Simpli-squared: Optimizing without cardinality estimates", in *SiMoD '24: Proceedings of the 2nd Workshop on Simplicity in Management of Data*, 2024, pp. 1–10.

[3] A. Hertzschuch, C. Hartmann, D. Habich, and W. Lehner, "Simplicity done right for join ordering", in *Proc. Conference on Innovative Data Systems Research (CIDR)*, 2021.

[4] G. Moerkotte and T. Neumann, "Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy trees without cross products", in *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2006, pp. 930–941.

[5] G. Lohman, "Heuristic method for joining relational database tables", *IBM Technical Disclosure Bulletin*, vol. 30, no. 9, pp. 8–10, 1988.

[6] L. Fegaras, "Optimizing large OODB queries", in *Proc. Int. Conf. on Deductive and Object-Oriented Databases (DOOD)*, 1997, pp. 421–422.

[7] L. Fegaras, "A new heuristic for optimizing large queries", in *Int. Conf. on Database and Expert Systems Applications (DEXA)*, 1998, pp. 726–735.

[8] V. Leis *et al.*, "Query optimization through the looking glass, and what we found running the join order benchmark", *VLDB Journal*, vol. 27, pp. 643–668, 2018.

[9] G. Moerkotte, T. Neumann, and G. Steidl, "Preventing bad plans by bounding the impact of cardinality estimation errors", *Proc. of the VLDB Endowment (PVLDB)*, vol. 2, no. 1, pp. 982–993, 2009.

[10] G. Cormode, M. Garofalakis, P. Haas, and C. Jermaine, *Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches* (Foundations and trends in databases). NOW Press, 2011, vol. 4:1–3, pp. 1–294.

[11] F. Olken and D. Rotem, "Simple random sampling from relational databases", in *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 1986, pp. 160–169.

[12] F. Olken, "Random sampling from databases", Ph.D. dissertation, U. California at Berkeley, 1993.

[13] 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 *Proc. of the ACM SIGMOD Conf. on Management of Data*, 1979, pp. 23–34.

[14] D. Flachs, M. Müller, and G. Moerkotte, "The 3D Hash Join: Building On Non-Unique Join Attributes", in *Proc. Conference on Innovative Data Systems Research (CIDR)*, 2022.

[15] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry, "Improving hash join performance through prefetching", *ACM Transactions on Database Systems*, vol. 32, no. 3, 2007.

[16] O. Kocberber, B. Falsafi, and B. Grot, "Asynchronous memory access chaining", *Proc. of the VLDB Endowment (PVLDB)*, vol. 9, no. 4, pp. 252–263, 2015.

[17] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer Manual, Vol. 3B*, available at https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, Jun. 2024.

[18] S. Setzer, G. Steidl, T. Teuber, and G. Moerkotte, "Approximation related to quotient functionals", *Journal of Approximation Theory*, Special Issue: Bommerholz Proceedings, vol. 162, no. 3, pp. 545–558, 2010.

[19] G. Moerkotte, "Building query compilers", available at https://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf, 2024.

[20] S. Cluet and G. Moerkotte, "On the complexity of generating optimal left-deep processing trees with cross products", in *Proc. Int. Conf. on Database Theory (ICDT)*, 1995, pp. 54–67.

[21] V. Leis *et al.*, "How good are query optimizers, really?", *Proc. of the VLDB Endowment (PVLDB)*, vol. 9, no. 3, 2015.

[22] G. Sun J. Li, "An end-to-end learning-based cost estimator", *Proc. VLDB Endow.*, vol. 13, no. 3, pp. 307–319, 2019. DOI: 10.14778/3368289.3368296.

[23] I. Trummer, "Exact cardinality query optimization with bounded execution cost", in *Proc. of the ACM SIGMOD Conf. on Management of Data*, 2019.

[24] R. Marcus *et al.*, "Neo: A learned query optimizer", *Proc. VLDB Endow.*, vol. 12, no. 11, pp. 1705–1718, 2019.

[25] R. Bergmann, A. Hertzschuch, C. Hartmann, D. Habich, and W. Lehner, "PostBOUND: PostgreSQL with upper bound SPJ query optimization", in *Proc. Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW)*, 2023.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org