The Selection Problem in Multi-Query Optimization: a Comprehensive Survey

Sergey Zinchenko, Denis Ponomaryov

ABSTRACT

View materialization, index selection, and plan caching are well-known techniques to speed up query processing in database systems. In all these tasks it is necessary to select and save a subset of the most useful candidates (views/indexes/plans) for re-use within a given space/time budget. In this paper, we propose a unified view on these selection problems, identify the root causes of their complexity, and provide a detailed analysis of techniques to cope with them. Our study gives a modern classification of selection algorithms known in the literature, including the latest ones based on Machine Learning. We provide a ground for re-use of the selection techniques between different optimization scenarios and highlight challenges and promising directions in the field.

1 INTRODUCTION

With the growing data storage and analysis demands, *Data Warehouses (DWH)* became increasingly widespread providing an unified access to data coming from a large number of heterogeneous sources. To mitigate the costs of configuring, maintatining, and scaling DB systems, platforms based on *Database-as-a-Service* (*DBaaS*) are now being widely implemented. Due to a typically large number of similar requests to service-based DB systems it proves useful to optimize incoming queries in series and reuse common computations between them. *Multi-Query Optimization (MQO)* aims at finding and reusing common computations for a more efficient workload execution. Savings achieved by re-use are typically called *benefit*. In general, the task is to find candidate computations for re-use which provide the highest benefit, while respecting the constraints on the available resources (e.g., disk space or computing time). This task can be divided into three *orthogonal* subproblems: 1) discovering common computations between queries, 2) *selecting* the most useful ones, and 3) making an optimal plan for their re-use. In this paper, we focus only on the second problem, i.e., the problem of selection.

One obtains different instances of this problem depending on the type of common candidate computations considered and the range of possible actions over the selected candidates. For example, in DWH scenarios, free disk space can be used to save (*materialize*) common data (*views*) [59]. The precomputed data can then be read from the disk instead of computing from scratch, which can speed up query execution by several orders of magnitude. The selection problem in this scenario is typically called *View Selection Problem (VSP)*, which is to identify a set of views that gives the highest benefit for a workload and fits the storage and maintenance budgets. Execution of a workload can also be accelerated by creating indexes that make access to data faster. In general, the *Index Selection Problem (ISP)* is similar to VSP because index can be considered as a special case of a single-table, projection-only materialized view [1]. MQO is also employed in the context of stored procedures and analytical reports. Due to the relatively small data and high processing time for reports, the latency of report generation can be reduced by storing candidate computations for re-use in the memory. In the literature, the selection problem for this scenario is referred to as *Query (Result) Caching* and it is very similar to VSP. Another way to speed up queries is to reduce planning time. This can be achieved by caching good plans and by reusing them for similar queries. The problem of selecting the most useful plans in such scenarios is known as *Plan Caching*.

As we will show, instances of the selection problem have certain specifics in different scenarios. In this paper we note however that **selection algorithms are agnostic to the nature of candidates they manipulate with[1], and thus, they can be reused between optimization scenarios under similar constraints**. Motivated by this observation, we formulate a generalized Candidate Selection Problem that abstracts away from the nature of candidates: they can be views, indexes, cached data, or even plans.

Our contribution can be summarized as follows:

(1) we make a detailed analysis of the root causes of the complexity of selection problems and summarize techniques to cope with them
(2) based on the View Selection Problem, we introduce a modern classification of selection algorithms, including the recent ones based on Machine Learning
(3) we propose a general framework of Candidate Selection which allows for reusing ideas and techniques between different instances of selection problems including View/Index Selection and Query/Plan Caching
(4) we highlight challenges, open questions, and promising directions in the development of selection algorithms for Multi-Query Optimization

The paper is organized as follows. In Section 2, we review related literature including surveys on similar topics and highlight the main differences with our work. In Section 3, we begin our study with examples of the main issues related to Selection and formulate the general Candidate Selection Problem. Techniques to resolve these issues are introduced in Section 4. We continue our analysis in Section 5 with a classification of selection algorithms, emphasizing the main techniques that can be (re)used to solve the Candidate Selection Problem (and thus, instances thereof). Finally, in Section 6 we describe open questions, challenges, and promising directions for future research and in Section 7 we conclude.

[1] only the way objects are represented and benefits are computed is important

2 RELATED WORK

Most related to our work is the survey paper [40] from 2012 which provides a classification of algorithms for View Selection. Recently some novel algorithms have been proposed, which present a new line of research related to Machine Learning. We review these algorithms in detail in our paper. Also, unlike [40], we make a detailed analysis of issues behind the proposed algorithms and explain the reasons for the design decisions made for them. We believe that this study could facilitate the development of better solutions. We also propose a general framework based on the Candidate Selection Problem, via which the previously proposed algorithms can be used interchangeably in different selection scenarios.

We now list the topics that are intentionally not covered in this paper. One of the very first surveys on View Selection was focused on the problem of choosing a view definition language and self-maintaining a set of views [18]. This topic, as well as the problem of finding an optimal way to use materialized views (*Query Answering Using Views* [24]), is relevant for building efficient solutions, but it is not focused on the selection itself and thus, not addressed in our paper. Our exposition does not cover techniques from Data Mining [52], Constraint Programming [41], and Game Theory [2], which did not attract much interest in the context of selection problems. Also, we do not consider strategies and techniques for updating selected candidates, as well as the relationship of the selection problem to the topics such as stream data processing, approximate query processing, etc. We recommend [8] as a starting point for exploring these fields. For an introduction to the Index Selection Problem, we recommend [4]. We also do not touch questions of coupling computation caching with other techniques such as query execution scheduling and pipelining, which are studied in paper [14]. We also mention that details on the problem of Plan Caching for a template of parameterized queries can be found in [16, 27, 28, 54] In the literature, these topics are referred to as *Parametric Query Optimization.*

3 PRELIMINARIES

We begin our exposition with an analysis of the principal issues related to selection problems. We introduce the necessary terminology and then formulate the Candidate Selection Problem as a unified view on the selection tasks. Then we summarize results related to the computational complexity of this problem.

3.1 Representation of Candidates

Let Q be a workload (a set of queries). Assume that besides the base relations in the database we have some precomputed information (e.g., views, indexes, or cached query plans) denoted as C, which we can use for processing these queries. An execution plan for an individual query $q \in Q$ generally depends on C; we denote it as $\mathcal{P}_C(\{q\})$. The union of plans for all $q \in Q$ is called *execution plan for workload* Q and it is denoted as $\mathcal{P}_C(Q)$.

The Multi-Query Optimization Problem (MQO) is to compute a set C such that $\mathcal{P}_C(Q)$ is optimal for executing Q. This problem is fundamentally more complex than individual query optimization,

[2]it is assumed that the read time of the data is the same as its size, and the data is permanently saved to disk between executions of the operations

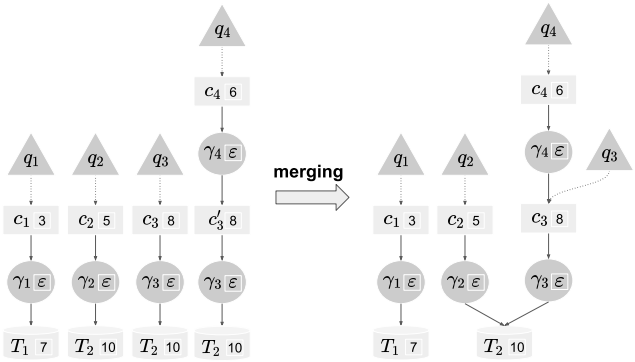

Figure 1: On the left: representation of the workload in the form of Expression Trees. On the right: the result of merging them into a Expression Forest $\mathcal{F}$. Eq-nodes (c_i) are given in rectangles, op-nodes (γ_i) are given in circles. Data size and the latency of operations are given in small boxes inside these figures[2]. Triangle-shaped nodes are used to depict which data every query q_i needs.

because it introduces the option of reusing items from C between queries, as shown below on the example of VSP:

Example 3.1. (Ex. 1.1 from [49]) Let workload Q consist of queries $q_1 = T_1 \bowtie T_2 \bowtie T_3$, $q_2 = T_2 \bowtie T_3 \bowtie T_4$ and let $\mathcal{P}(\{q_1\}) = (T_1 \bowtie T_2) \bowtie T_3$ and $\mathcal{P}(\{q_2\}) = (T_2 \bowtie T_3) \bowtie T_4$ be their *individually* optimal plans, respectively. Although these plans are optimal for each query separately, it can be the case that by *reusing* $C = \{T_2 \bowtie T_3\}$ one obtains an optimal plan for the entire Q. That is, using the join sequences $T_1 \bowtie (T_2 \bowtie T_3)$ and $(T_2 \bowtie T_3) \bowtie T_4$ gives a total latency for both q_1 and q_2 lower than the sum of the latencies obtained by using $\mathcal{P}(\{q_1\})$ and $\mathcal{P}(\{q_2\})$.

In MQO, each query is typically represented in the form of an *expression tree* which is built according to its execution plan. An expression tree is a directed bipartite acyclic graph, which represents required data, operations over it, and the result.

Definition 3.2. An expression tree is a pair $\langle V_{op} \sqcup V_{eq}, E \rangle$, where V_{op} is a set of operation nodes (op-nodes, for short), V_{eq} is a set of data nodes (known in the literature as equivalence nodes or eq-nodes, for short[3]), and E is a set of directed edges, which can be of two types. The first edge type ($v_{eq} \rightarrow v_{op}$) indicates that operation v_{op} is applied to data v_{eq}. The second type of edge ($v_{op} \rightarrow v_{eq}$) indicates that v_{op} must be performed to retrieve data v_{eq}.

Then expression trees obtained for individual queries are joined into an *expression forest* by merging common nodes.

Example 3.3. Let workload Q consist of the following four queries, the optimal plans $\mathcal{P}(\{q_1\}), \ldots, \mathcal{P}(\{q_4\})$ for which are shown on the left in Fig. 1.

```
q1: SELECT week, AVG(price) as avg_price
    FROM T1 GROUP BY week;
q2: SELECT week, AVG(price) as avg_price
    FROM T2 GROUP BY week;
```

[3]due to the fact that for each date node there can exist several computation paths, which give equivalent results

```
q3: SELECT week, day, AVG(price) as avg_price
    FROM T2 GROUP BY week, day;
q4: SELECT week, AVG(avg_price) as macro_avg_price
    FROM (
        SELECT week, day, AVG(price) as avg_price
        FROM T2 GROUP BY week, day
    ) as weekly_price
    GROUP BY week;
```

In the resulting expression forest, plans $\mathcal{P}(\{q_2\}), \mathcal{P}(\{q_3\})$ and $\mathcal{P}(\{q_4\})$ are merged due to the common read of table T_2. Also, since the aggregation by $\langle week, day \rangle$ in queries q_3 and q_4 is the same, the data of c_3 and c_3' is the same and these nodes are merged too.

Candidates for re-use are searched among eq-nodes of the resulting expression forest. As each eq-node corresponds to the result of the execution of some subexpression, the search space in this approach is also called *subexpression space*.

This simple approach has a significant drawback. Since an expression forest is built over optimal plans for individual queries, some candidates for building a plan optimal for the entire workload can be *missed* (as already shown in Example 3.1). In order to obtain an optimal solution, one needs to represent a query as an expression tree in a way that *takes into account alternative computation paths*. To achieve this, one can use graphs of a special type, which we consider further in Section 4.1. However, having all possible expression trees available may not suffice in general. As shown by Example 1 in [7], the most optimal solution may contain a candidate which *is not a subexpression* of any of the queries in a workload even if we consider *all alternatives*. These observations indicate that choosing an appropriate **representation of candidates** is an essential step in solving the selection problem.

3.2 Benefit of Candidates

Typically, the total benefit $\mathfrak{B}(C)$ of a subset C of candidates for a workload Q is defined as $T_\varnothing(Q) - T_C(Q)$ where, for a set S, $T_S(Q)$ means the execution time of Q with plan $\mathcal{P}_S(Q)$. It is natural to define *benefit* $\mathcal{B}_c$ of a candidate as the savings of the execution time obtained by reusing c in the optimal plan $\mathcal{P}_C(Q)$. Note that benefit $\mathcal{B}_c$ depends on the other elements in C, because an optimal plan depends on the whole set of candidates available for reuse:

Example 3.4. Consider the workload from Example 3.3 depicted in Figure 1. Suppose that the data retrieval process consists of a) reading all operands relevant to the corresponding operation and b) executing it. Then in a plan $\mathcal{P}_{\{c_3\}}(Q)$ we use c_3 to answer *both* queries q_3, q_4 and thus, the benefit $\mathcal{B}_{c_3}(\{c_3\})$ equals $2 \cdot [(10+\varepsilon)-8] = 2 \cdot (2+\varepsilon)$. Indeed, instead of reading table T_2 and executing γ_3 in $10+\varepsilon$ time, we just read c_3 *twice*, each time spending 8 units of time. However, in plan $\mathcal{P}_{\{c_3,c_4\}}(Q)$ the benefit $\mathcal{B}_{c_3}(\{c_3, c_4\})$ equals $2+\varepsilon$, as we use c_3 only to execute q_3. This demonstrates that the benefit of c_3 *is decreased* when c_4 is selected.

Another problem is that estimates for operation latency and data size for nodes in an expression forest can be *inaccurate*, since they are typically obtained from a database optimizer. Moreover the estimation errors are multiplied if the same operation is used on several computation paths simultaneously. Since it is problematic to obtain an optimal solution with wrong estimates, the problem of accurate **benefit estimation** becomes crucial. We consider different ways to approach this problem in Section 4.2.

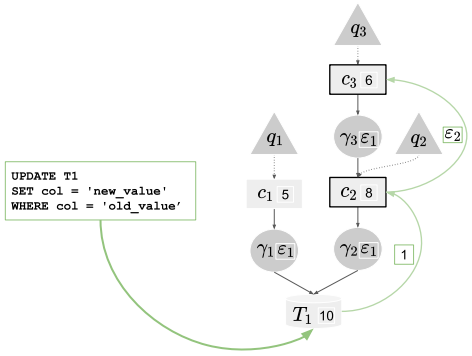

Figure 2: A workload with the selected candidates shown in black rectangles. When T_1 is updated, the candidates *must* also be updated. Update operations with their execution times are shown in green.

3.3 Constraints

It can be shown that the total benefit does not decrease with the expansion of the set C of candidates, so one potentially obtains the highest benefit when all computations are selected for reuse. Obviously, this is impractical, since selecting a candidate c incurs some non-zero *expense* e_c which is related, e.g., to the disk space used for storing c. Similar to benefit, the expense of selecting a candidate depends on other elements in C. For example, in VSP, it is necessary to keep the data in materialized views up-to-date, so e_c represents the time required to update a view c [21]. Given that other views may be used for the update, the expense e_c depends on other candidates in C:

Example 3.5. Consider the workload from Figure 2. If c_2 is selected, the time of updating c_3 will be shorter, since we can re-use the *updated* c_2:

$$\underbrace{8+\varepsilon_2}_{\text{update } c_3 \text{ over } c_2} \quad \text{vs} \quad \underbrace{10+\varepsilon_1}_{\text{calculate } c_2} + \underbrace{1}_{\text{update } c_2} + \underbrace{8+\varepsilon_2}_{\text{update } c_3 \text{ over } c_2}.$$

The non-linear behavior of expense occurs also in *Plan Caching*, in which the problem is to optimally reuse computed plan trees (*not the data itself*). Frequently occurring subtrees can be stored separately and re-used when necessary (see Example 3.6). Then the expense of storing a plan depends on whether *any of its subplans is already in the cache*. As shown in [11], Plan Caching can improve performance, and the marked behavior of e_c can help enhance it:

Example 3.6. Let the optimal plans of some queries q_i, q_j from a workload Q have a common subtree ST (see Figure 3). Instead of saving ST for each of the plans $\mathcal{P}(\{q\})$ we can save it *once*, and refer to it in all its supertrees. Then the saved cache space can be used to speed up Q by caching additional plans.

In general, one has to deal with with the constraint $\sum_{c \in C} e_c \leq E$, where E is an expense budget, which represents, e.g, the available

[4] the set of edges depicted with an arc is the and-arc (see Section 4.1 for more details)

parameter \ problem	View Selection	Index Selection	Query Caching	Plan Caching
candidate space $\mathbb{C}$	eq-nodes	eq-nodes	eq-nodes	**subtrees**
expense $e_c(C)$	non-constant[a]	non-constant[a]	constant	non-constant[b]
benefit $\mathcal{B}_c(C)$	non-constant, computed as the speed-up under reuse of c in plan $\mathcal{P}_C(Q)$			

[a] under maintenance constraint
[b] when compressed tree storage is used

Table 1: Instances of the Candidate Selection Problem

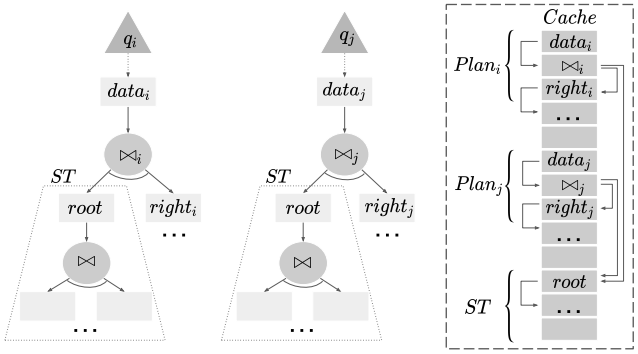

Figure 3: On the left are the optimal execution plans for some queries q_i, with a common subtree ST[4]. On the right is a plan caching schema, in which the plan for ST is stored in memory only once.

storage space or time for update. As we will show further in the paper, the problem of the **non-linearity of expense** fundamentally complicates the selection problem.

3.4 Candidate Selection Problem

The examples above demonstrate that in all the types of selection problems considered the nature of candidates is not important: it is only relevant how the benefits and expenses are computed. Therefore, we now introduce a generalized *Candidate Selection Problem (CSP* [5]) which highlights the main aspects of the selection tasks.

Let $\mathbb{C}$ be a set of *candidates* and let $\mathcal{B}_c(\cdot)$ and $e_c(\cdot)$ be functions which for every candidate $c \in \mathbb{C}$ give its benefit and expense, respectively. The Candidate Selection Problem is to select a subset of candidates $C \subseteq \mathbb{C}$ which gives the maximal total benefit $\mathfrak{B}(C)$ under a given expense budget E:

$$\mathfrak{B}(C) := \sum_{c \in C} \mathcal{B}_c(C) \rightarrow \max_{\{C \subseteq \mathbb{C} \mid \sum_{c \in C} e_c(C) \leq E\}} \quad (1)$$

CSP **represents all selection problems in Multi-Query Optimization** (see Table 1). It is worth noting that, unlike benefit (which is non-linear in all the selection problems considered), the behavior of $e_c(\cdot)$ function depends on the type of the problem.

Definition 3.7. CSP is *monotone for benefit* if for any candidate set $C = C_1 \sqcup C_2$ it holds that $\mathfrak{B}(C_1 \sqcup C_2) \leq \mathfrak{B}(C_1) + \mathfrak{B}(C_2)$ and it is *monotone for benefit per unit space* if $\rho(C_1 \sqcup C_2) \leq \rho(C_1) + \rho(C_2)$, where $\rho(C) = \frac{\mathfrak{B}(C)}{\mathcal{E}(C)}$ and $\mathcal{E}(C) = \sum_{c \in C} e_c(C)$.

We will show later how the monotonicity property relates to the complexity of the selection problem.

3.5 Computational Complexity

In this section, we demonstrate the hardness of the Candidate Selection Problem by giving an overview of the complexity results on the View Selection Problem (which is clearly, an instance of CSP).

It is known that under the space constraint VSP is NP-hard. It has been also proved that a greedy algorithm can give a solution that is at least 63% of the optimal one (47%, respectively, for the case when views are selected together with indexes), and this lower bound *can not be improved* in polynomial time [20]. Under the *maintenance constraint* (a limited budget for keeping views up-to-date) there is no similar result: solutions obtained greedily *can be arbitrarily bad* [21]. The reason is the lack of the monotonicity of the benefit per unit space due to the non-linear behavior of e_c. The drop in accuracy from 63% to 47% in the case of selecting views together with indexes is also due to the lack of monotonicity, but the reason is the non-linear behavior of the benefit itself. Indeed, we can only benefit from the index if the corresponding view is selected.

In minimizing the total execution time, VSP is known to be *polynomially non-approximable* [33]. This does not contradict the result with the 63% accuracy of a greedy algorithm, because by using the benefit we implicitly transition to a closely related (i.e., their optimal solutions coincide) but still a different optimization problem. The original optimization objective is the total time $T_C(Q)$, but when using benefits we are maximizing $\mathfrak{B}(C) = T_\varnothing(Q) - T_C(Q)$. Let $\widehat{C}$ and C^* denote the obtained and the optimal sets of candidates, respectively. Then clearly, from the relationship $\mathfrak{B}(\widehat{C}) \geq k \cdot \mathfrak{B}(C^*)$ it is impossible to derive the total time in terms of k. The reason is that the ratio of $T_\varnothing(Q)$ to $T_{C^*}(Q)$ is unknown:

$$\left(\frac{\mathfrak{B}(\widehat{C})}{\mathfrak{B}(C^*)} = \frac{T_\varnothing(Q) - T_{\widehat{C}}(Q)}{T_\varnothing(Q) - T_{C^*}(Q)}\right) = k \Leftrightarrow \frac{T_{\widehat{C}(Q)}}{T_{C^*}(Q)} = k + (1-k) \cdot \frac{T_\varnothing(Q)}{T_{C^*}(Q)}$$

For the case when query plans contain non-unary operators (e.g., joins), approximability *is still an open question* for VSP even in the benefit setting under the space constraint (as well as the general question of the computational complexity) [22]. Besides, if we consider only plan nodes as candidates (no matter how they are represented), we may miss the optimal solution. Chirkova et al. [7]

[5] not to confuse with the Constraint Satisfaction Problem

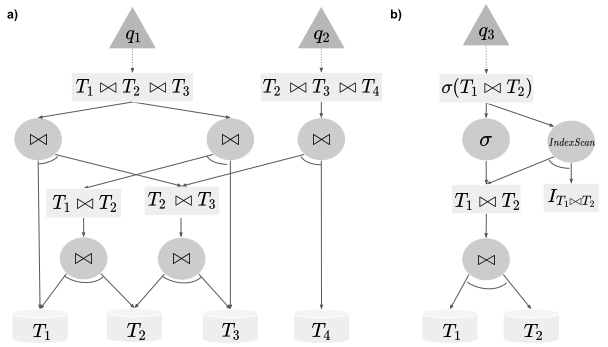

Figure 4: a) AND-OR-DAG representing two alternative ways of executing query $T_1 \bowtie T_2 \bowtie T_3$ and a single way of executing $T_2 \bowtie T_3 \bowtie T_4$. The option of reusing $T_2 \bowtie T_3$ is shown. b) Index utilization plan in which the the need for both, an index and a materialized view is indicated by an and-arc.

showed that, in general, the space of candidates that needs to be considered to find an optimal solution is *infinite*.

To overcome the **algorithmic complexity** of the selection problem, a plethora of approaches has been proposed including heuristic-based, randomized algorithms, or custom ones which provide solutions close to the optimum only for a fixed pool of queries and specific underlying data. The latter class of algorithms is based on the recent Machine Learning approaches. There is also a number of optimizations to reduce the search space and employ the tree structure when computing benefits and expenses. We will discuss these techniques in more detail in the next two sections.

4 PREPARATION FOR SELECTION

We now highlight several important techniques that are used prior to the selection stage to make the overall procedure more efficient.

4.1 Query Representation

Subexpression Space. Although an optimal solution can be missed when *only* subexpressions of queries are used for building a space of candidates, this approach still has several important advantages. First, it makes the process of building the candidate space relatively simple: candidates are searched only among the nodes of the plan obtained from the optimizer and alternative computation paths are avoided, which greatly reduces the search space. Second, since all candidates are subexpressions, their execution statistics can be used to approximate benefits. Alternatively, special techniques are sometimes used to modify subexpressions and build new candidates given the specific workload and the nature of the MQO problem, which makes the search space richer. We discuss this in more detail at the end of this subsection.

Representation Frameworks. Even when the candidate space is given by subexpressions there is still a freedom in choosing a representation for expression trees. For example, in order to account for the existence of alternative execution paths, an expression tree can be represented as a *OR-DAG*. This is equivalent to allowing eq-nodes to have multiple children, which correspond to different computation paths. To represent non-unary operations (e.g., joins), the *AND-DAG* representation framework can be used, which introduces *and-arcs* as several directed edges connected by an arc. This representation takes into account the need to compute *all* operands to execute an operation. Definitions of these classes of DAGs can be found in [22]. This work also introduces a notion of *AND-OR-DAG*, which combines the features of the two previous frameworks.

Example 4.1. As one can see in Example 3.1, the option of using $T_2 \bowtie T_3$ for both queries must be available. This can be implemented by representing multiple computation paths for node $T_1 \bowtie T_2 \bowtie T_3$ with an or-arc (see Figure 4a). The arcs under joins are and-arcs, which indicate the need compute *all* operands for a join operation. Part b) of the figure shows that and-arc can represent a plan which uses an index: to use the index (op-node *IndexScan*) we need both, the data (eq-node $T_1 \bowtie T_2$) and the index itself (eq-node $I_{T_1 \bowtie T_2}$).

In OLAP scenarios, queries often refer to table aggregations and are uniquely defined by a set of columns in a GROUP BY clause. Hence, it is possible to represent these queries as vertices of the hypercube given by the set of table attributes. The 'can-be-computed-by' relation over queries then coincides with the inclusion relation over the sets in the hypercube, so this representation is called *Hypercube Lattice* (or *Data Cube*). If a workload Q is represented in the the form of a Data Cube, there is *no need to build an expression forest*, since all query relationships are *already given* by the cube. However the Data Cube introduces a space of 2^n vertices (where n is the number of table attributes) in which all possible aggregate queries to the table are represented. If aggregation is ranked according to some granularity (e.g., by day, week, or year), one also has to consider the *granularity hierarchy*. For example, the *Product Graph* (*direct product* in [25]) framework provides this feature.

Choosing a representation framework is an important step, because the complexity of selection strongly depends on the way query plans are represented (see Section 3.5).

Computation Sharing. As we have already noted, the set of nodes of an expression forest may not contain an optimal candidate. To deal with this problem, one can use additional techniques to enrich the search space. For example, one can employ the technique of additional computations:

Example 4.2. Consider a workload of two queries $\sigma_A(T_1)$ and $\sigma_B(T_1)$ given in Figure 5. Suppose, we have a space budget, $E = 9$, which is insufficient to store the answers to both queries ($E = 9 < 6 + 6 = e_{c_1} + e_{c_2}$). Then we can generate and store an *additional computation* $c_3 = \sigma_{A \vee B}(T_1)$ that can be shared between the two queries. This gives the maximal possible benefit

$$\mathcal{B}(c_3) \approx 2 \cdot (\underbrace{10}_{\text{read } T_1} - \underbrace{7}_{\text{read } c_3}) = 6$$

(compared to the cases when c_1 or c_2 are stored), but implies the need of *extra* operation, (termed as *compensations* in [9]) for fetching the required data. For example, to answer query q_1, we must apply extra selection σ_A to candidate c_3, which, in turn, was also obtained using selection $\sigma_{A \vee B}$.

Also various 'rewriting' strategies based on relational properties [61] and identities [48, 49] can be used. By choosing rewrite rules

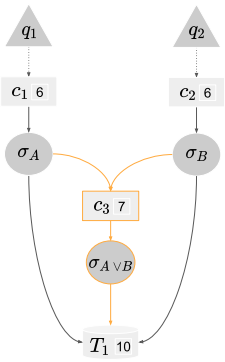

Figure 5: Illustration of the technique of introducing additional computations (which are shown in orange).

and the order of their application to the nodes of individual query plans one can influence the number of common nodes among plans, as well as their sizes. This may have a positive impact on the quality of the resulting solution. We provide more details on this technique in Section 5.2.3 where we describe the corresponding algorithms.

4.2 Benefit Estimation

Lightest Path and Cost Models. In the expression tree framework, it is easy to see that query latency is determined by the weight of the lightest computation path from the eq-node corresponding to the query to the leaves[6] of the tree. To compute the weight of a computation path we need to take card of arc types: 1) if an or-arc encountered, we can continue the computation path through *any* of the children, 2) for a and-arc, we have to compute *all* the children. A *useful property* of tree-calculated benefits is that the time of executing a node c is affected only by the selection of its descendants. But interestingly, the benefit of a candidate c depends on its parents too, since their selection may change the lightest computation path preventing c from being used (see Example 3.4). In Section 5.2, we discuss how to employ this observation.

To compute the weight of a computation path, one also needs to know the estimates for operation latencies and data sizes, which can be obtained from the optimizer. To combat the inaccuracy of these estimates, predictive models have been proposed, which, use, e.g., run-time statistics on the execution of queries for making predictions about the future [31].

End2End Modeling. Instead of representing queries as trees and searching for the lightest computation paths one can use the optimizer in *what-if* mode [68]. The idea is to simulate the situation when a candidate computation of interest is reused. Then the candidate benefit is the difference between the optimizer's predicted values with and without the candidate. Since the benefit behaves non-linearly we have to compute it by a call to the optimizer each time the set of candidates C changes, which is inefficient. Instead, one can train a ML model to answer questions like "what is the benefit $\mathcal{B}_c$ of a candidate c for a given workload Q if we have already selected candidates C?" [63].

[6] or saved eq-nodes, because we can read them instead of executing from scratch

IP. There are also approaches in which the problem of node selection and re-use is reduced to an Integer Programming problem [60]. This is a simple way to obtain an exact solution for the selection problem (provided accurate estimates are available), from which it is also easy to compute the benefit $\mathcal{B}_c$ of each candidate c. Indeed, it can be obtained as the difference between the execution time of c and the time of its reuse, multiplied by the number of uses of c in the optimal solution. We note however that computing an *exact* solution to an IP problem to obtain benefits makes little sense for the following reasons. First, if an optimal solution is found, then then benefits are not needed anymore. Second, in order to formulate the selection problem as an IP, one needs to introduce a large number of additional variables, which would make solution search prohibitively long. Instead, one can first fix a set of candidates C, and then solve simpler problems of finding the optimal way to use them *independently*, for each query q from a workload. These subtasks can be solved in parallel giving an approximation of benefits. In Section 5.4.2 we discuss several possible ways to implement this idea.

4.3 Dealing with Constraints

Specialized Solutions. Recall that the constraint in CSP is given as $\sum_{c \in C} e_c(C) \leq E$ and in some cases the expense function $e_c(\cdot)$ may be non-linear. That is, for $C = C_1 \sqcup C_2$ it holds in general that

$$\sum_{c \in C} e_c(C) \neq \sum_{c_1 \in C_1} e_{c_1}(C_1) + \sum_{c_2 \in C_2} e_{c_2}(C_2).$$

This poses challenges to selection algorithms: for example, a greedy algorithm can no longer guarantee any % of accuracy, in contrast to the case of space constraints. In part, this can be overcome by specialized solutions. For example, in [22] Gupta et al. made a theoretical analysis of the behavior of the greedy algorithm and proposed to use special *inverted set tree* structures to achieve the desired accuracy of 63%. However, the complexity of this solution is exponential in general.

Penalization. Even simple constraints with constant e_c can pose difficulties. For example, in randomized and genetic algorithms, the set of candidates is built iteratively, and at each step the action to be taken for a candidate c must depend on its benefit $\mathcal{B}_c$. The problem here is two-fold. First, the benefit takes into account neither the expense of the candidate, nor the remaining budget. Second, situations, in which a constraint is violated, must be handled accordingly. Simply avoiding such situations may result in a poor strategy. Indeed, the path which goes only through admissible solutions from the current solution to a good local optimum may either not exist or be very long. To address this, a penalization approach has been proposed [36], which tries to account for the presence of a constraint in the form of some penalty (*regularizer*). The simplest way to implement this is the *substract mode*. The idea is to use the penalty function $\phi(C) = \max(0, \sum_c e_c(C) - E)$ and compute the benefit as $\mathfrak{B}'(C) = \mathfrak{B}(C) - r \cdot \phi(C)$, where r is some regularization coefficient. The value of r affects "how much we don't want to violate the condition $\sum_{c \in C} e_c(C) \leq E$". Important is to choose the right coefficient r, because if it is chosen badly, the algorithm will tend to "non-violation of constraints" instead of optimization. This can be avoided by measuring everything in the same units. For example, in VSP, if we know the rates of

computational resources at runtime, as well as the price per unit of disk space, we can express everything in money and maximize the total profit [63]. This makes sense, because there is no problem with disk space and memory availability nowadays, the only issue is payback.

Stochastic Solutions. If a selection algorithm uses benefits only to compare candidates (as in the local search algorithms), we can use *stochastic ranking* instead of penalization [36]. In short, the idea is to employ a comparison that favors more cheap candidates with probability $1 - p$ with no regard to their benefits [50].

5 SELECTION ALGORITHMS

We analyse in detail the techniques sketched in the previous sections, we provide a modern classification of the selection algorithms employing these techniques, and discuss the main trends in the development of such algorithms. Our exposition is primarily based on an analysis of view selection algorithms, with the emphasis on the techniques that can be reused for solving various instances of the Candidate Selection Problem in Multi-Query Optimization.

5.1 Exhaustive Search

Assume a workload Q consists of queries that involve joins over different subsets of tables $\{T_j\}_{j=1}^{n}$. Then, in the worst case, the size of the search space (the number of candidates among which it makes sense to look for a solution) is 2^n (the number of all possible joins). As the benefit $\mathcal{B}_c$ of an individual candidate non-linearly depends on the set of selected candidates C, a naive search for an optimal selection would have to enumerate all possible subsets of C, which is already of double exponential size 2^{2^n}. This kind of algorithm was proposed in [47], but its complexity is prohibitive for the size of modern databases.

5.2 Heuristics

5.2.1 Optimizations of Greedy Algorithm. In many scenarios, it suffices just to compute a good enough solution within a constrained time budget. To implement this, one can search for a solution only in a certain subexpression space. For this, plenty of heuristics has been proposed which aim at selecting the most useful candidates [30]:

(1) **Topk-freq**: selecting the most common candidates
(2) **Topk-utility**: selecting candidates with the maximal benefit they can provide for an individual query
(3) **Topk-TotalUtility**: selecting candidates with the highest total benefit for entire workload
(4) **Topk-NormTotalUtility**: selecting candidates with the highest specific total benefit for entire workload per space unit

These heuristics do not take the non-linearity of benefit into account and thus, they can give poor solutions.

The paper [25] addresses this shortcoming. The authors propose a greedy algorithm in which, at every step, the currently selected set of candidates C is expanded with a candidate c, which gives the largest total benefit $\mathfrak{B}(C \cup c)$. It was proved in [33] that this algorithm guarantees the accuracy of at least $(1 - \frac{1}{exp}) = 63\%$, and no polynomial time algorithm can do better. We note however that the constraint considered in [25] is the number of selected candidates, not the space they occupy.

Gupta et. al in [20] adopted a similar idea for the case of the space constraint. They proposed to measure the benefit change, when a candidate c is added, per the unit of space it occupies:

$$b_c(C) := \frac{\mathfrak{B}(C \cup c) - \mathfrak{B}(C)}{\mathcal{E}(C \cup c) - \mathcal{E}(C)},$$

They proposed an algorithm for the OR-DAG and AND-DAG frameworks which in both cases provided the accuracy guarantee of 63%. For the general AND-OR-DAG framework, the authors proposed a *AO-Greedy* algorithm which is in fact a greedy algorithm over specially defined *intersection graph* structures. The size of this structure is exponential in general, so the proposed algorithm is no longer polynomial in the size of the expression forest.

For the setting when indexes are selected together with views, Gupta et al. proposed an *inner-level greedy algorithm*. The option of selecting indexes breaks the monotonicity property, because one can not use an index if the corresponding data is not selected (see Example 4.1), and as a consequence, the benefit of this index is zero. The inner-level greedy algorithm guarantees a 47% accuracy in OR/AND-DAG frameworks and it is based roughly on the following idea. At each iteration, the algorithm first searches for a view which has the highest benefit along with its best indices. Then the set of candidates C is extended either by this view and its indices, or by one new index (probably, for another view) having the highest benefit. This procedure allows one to avoid the following problem: if a view is useful *only* with an index, then it *can not be selected* by the basic greedy algorithms. Indeed, the index will not be selected, because without the view its benefit is 0, and the view is not is selected because without an index its benefit insufficient. To adapt their method to the AND-OR-DAG framework, the authors proposed a generalization in the form of a *r-level greedy* algorithm, which is able to guarantee a certain % of accuracy in situations with more complex dependencies in benefit.

For the setting with the maintenance constraint in the OR-DAG framework, the authors proposed to use an *inverted-tree set* structure to regain the monotonicity property. An estimate of 63% accuracy for a greedy algorithm using this set structure is guaranteed, but the number of these sets is exponential in the worst case.

In [44] two optimizations of the greedy approach were proposed by taking into account the tree structure in benefit computation. Since at each step the greedy algorithm searches for the best candidate c to expand C with, one needs to frequently evaluate the total benefit of sets that differ just in a single element. Since the benefit of the nodes is often computed by tree traversal, one can *cache benefits for nodes* and recompute them only when the choice of a new candidate c has an impact on them. Along with this technique, the authors proposed a *coarse heuristic* that assumes that the node benefit can only decrease[7]. With this heuristic, if the nodes are stored in the descending order of (the previously computed) benefits, then iteration over all candidate nodes outside of C can often be avoided in finding the best next candidate to expand C.

[7] this is not true in general, since there is also a non-linear update time l_c component in the overall execution time, which may decrease as new candidates are added

5.2.2 Candidate Space Reduction. Clearly, the running time of a selection algorithm depends not only on the selection procedure itself, but also on the size of the search space. Several approaches to reduce the search space have been proposed in the literature. In [3], each individual query plan is traversed in the *level-order*, and only vertices from the level that gives the highest benefit are taken into the candidate space $\mathbb{C}$. This also allows one to get rid of the complex dependencies in benefits and expenses as it is guaranteed that the parents and children of the candidates will not be selected.

In [1], the authors suggested to consider only those candidates which refer to the "most interesting" tables. A table T_1 is considered to be more interesting than a table T_2 if the total execution time of queries touching T_1 is higher than that of queries related to T_2. Based on the selected set of tables, the authors define the candidate space according to the following idea: "*if a candidate is not in the optimal plan of any of the queries, it is unlikely to be useful*". For each query q from a given workload they define a candidate space $\mathbb{C}_q$ as the union of all subsets of the most interesting tables touched by q (the authors also analyzed conditions in queries and formulated conditions for grouping or selection over joins of interesting tables). Then, with the help of a *greedy(m, k)* algorithm[8] they select from them the most interesting candidates C_q for each query q. The candidate space $\mathbb{C}$ for the selection problem is defined as the union of all these interesting candidates.

In this approach, the selection of the candidate space is based on optimizing queries individually. Therefore, the authors introduced a *MergeViewPair* algorithm to take into account the nature of MQO. The algorithm iteratively merges pairs of candidates c_1, c_2 into a single candidate c, while a) preserving the possibility of using c instead of c_1 and c_2, and b) guaranteeing a small computational overhead of using c instead of c_1 and c_2. It is worth noting that using c is often slightly more expensive, since it contains data from (at least) both c_1 and c_2. But the number of situations in which c can be reused is much larger, which is important for optimization of the entire workload (Example 4.2 illustrates this technique). At the final stage, *greedy(m, k)* selection algorithm is applied to the candidate set $\mathbb{C}$ obtained by merging.

Gupta et al. [22] proposed a slightly different approach based on the following idea: instead of reducing the entire search space apriori, we can avoid exploring those parts of the space where there is definitely no better solution. To implement this, the authors adapted the ideas of A^* algorithm [46]. They considered the situation when all eq-nodes of the expression forest must be computed ($\mathbb{C} = Q$). Their A^*-like algorithm looks as follows. The search space is represented as a *search tree*, with vertices $v = \langle C, \mathbb{C}_{seen}\rangle$ representing information about visited candidates $\mathbb{C}_{seen} \subseteq \mathbb{C}$ and selected ones $C \subseteq \mathbb{C}_{seen}$. There is an edge $v \to v'$ if the vertex v' is obtained from v by adding a candidate c, i.e., $v' = \langle C \sqcup \{c\}, \mathbb{C}_{seen} \sqcup \{c\}\rangle$ or $v' = \langle C, \mathbb{C}_{seen} \sqcup \{c\}\rangle$. The goal is to reach a vertex $v^* = \langle C^*, \mathbb{C}\rangle$ in which the selected set of candidates C^* gives the minimal time of executing the workload.

The algorithm tries to find this vertex by exploring the search tree *only in the most promising directions*. To do this, it assigns to each vertex $v = \langle C, \mathbb{C}_{seen}\rangle$ an estimate of the minimum time of executing the entire workload $\mathbb{C}$, provided we saved only C for the execution of $\mathbb{C}_{seen}$. The principal problem is to get an accurate estimation, because it directly influences the extent to which the search space is reduced. This problem can be solved as follows. Assume the optimal way to expand the current set of selected candidates C is $C' \subseteq \mathbb{C} \setminus \mathbb{C}_{seen}$. Then, by taking into account the additivity of the execution time, the best execution time for the workload achievable from the current vertex v can be decomposed:

$$T_{C \sqcup C'}(\mathbb{C}) = T_{C \sqcup C'}(\mathbb{C}_{seen}) + T_{C \sqcup C'}(\mathbb{C} \setminus \mathbb{C}_{seen}).$$

The key point of the algorithm is the traversal of vertices $c \in \mathbb{C}$ in a *topological order*, which guarantees that the execution time of queries from $\mathbb{C}_{seen}$ remains unchanged under future extensions of C, i.e. $T_{C \sqcup C'}(\mathbb{C}_{seen}) = T_C(\mathbb{C}_{seen})$ (the reason for this is explained in Section 4.2). Then instead of estimating the value $T_{C \sqcup C'}(\mathbb{C})$ we can compute the lower estimate $\widehat{h}(C)$ *of a simpler value* which is the execution time of the rest of the workload $\mathbb{C} \setminus \mathbb{C}_{seen}$. To do this the authors use a separate greedy algorithm (see [22] for details). At each iteration, the A^*-like algorithm selects a vertex for which the predicted execution time $\widehat{h}(C) + T_C(\mathbb{C}_{seen})$ is minimal. This significantly reduces the search space in practice. The algorithm returns the exact solution, but in the worst case it has to explore the entire graph of size exponential in $\mathbb{C}$.

Labio et al. [35] employed similar ideas in a slightly different scenario. They considered a setting in which *already materialized* views Q need to be updated efficiently. One way to achieve this is to spend some resources on *materialization of new computations* which speed-up updates of Q. The problem can be formulated in terms of the Candidate Selection Problem as follows. For a given workload Q (queries that describe already materialized views), select a set of candidates C from an appropriate candidate space $\mathbb{C}$ such that a) the total update time for Q with C is minimal and b) $Q \subseteq C$. The latter requirement is important, since the views Q are already materialized Q and we have to spend resources on updating them. The authors adapted A^* algorithm for this scenario. In this setting, for vertices $\langle C, \mathbb{C}_{seen}\rangle$, it is required to estimate the minimum time of the total update after expanding C with a set C' provided $Q \subseteq (C' \sqcup C)$. The total update cost *(UC)* can obviously be divided into two parts:

$$UC_{C' \sqcup C}(C \sqcup C') = UC_{C' \sqcup C}(C) + UC_{C' \sqcup C}(C').$$

If the vertices are traversed in the topological order then the dependence of $UC_{C' \sqcup C}(C)$ on C' can be avoided. The remaining update cost $UC_{C' \sqcup C}(C')$ can be estimated by a separate greedy algorithm. In practice, with this approach the authors obtained a 4 orders of magnitude reduction of the search space.

5.2.3 Design of Candidate Space. Heuristic approaches are also used for a targeted design of a candidate space which should contain a high quality solution. For example, in [61] a method to build an expression forest $\mathcal{F}$ takes into account the fact that individually optimal plans may not contain the most useful computations for executing a given workload Q. First, an optimal plan $\mathcal{P}(\{q\})$ is built for every query $q \in Q$. Then the selection and projection operations are pushed up in expression trees, which makes the candidates larger and hence, implies potentially more savings from their re-use. Next, the resulting plans $\mathcal{P}'(\{q\})$ are merged into a

[8] greedy(m, k) algorithm first searches exhaustively for a subset of *good* k elements and then it expands this set in a greedy manner

forest $\mathcal{F}$ in a cost-based manner. Finally, all the select and project operations in $\mathcal{F}$ are pushed down[9], and epy greedy algorithm is run.

In [49], two more algorithms to design expression trees are proposed. In the first one, *Volcano-SH (SHared)*, individually optimal plans are first built independently and then the techniques of *computation sharing* described in Section 4.1 are applied. In the second one, *Volcano-RU (ReUse)*, plans are built sequentially. This makes possible to figure out which nodes belong to optimal plans for other queries. The execution time for these nodes is deliberately underestimated, so that they are more often found in optimal plans for other queries. Because of this the resulting expression tree has more common nodes and provides options for their reuse. Node benefits are computed in a special way. To calculate the benefit of reusing a candidate c one needs to know a) the time of computing c and b) the frequency $num\_uses(c)$ of using c. Therefore, the benefit depends on both the children and parents of a candidate node. The first dependency is eliminated by traversing vertices in the topological order and the second one by heuristically estimating $num\_uses(c)$ by the number of ancestors of c: $num\_uses(c) \geq \#ancestors(c)$ [10]. This approach allows for quickly and accurately estimating the benefit.

5.3 Randomized Algorithms

As we can see, all known heuristic methods are either rather time consuming (A^* algorithm) or they return an approximate solution (by greedy algorithms). We now consider randomized algorithms which provide an efficient alternative to heuristic ones.

5.3.1 Random Sampling. In [32], several randomized algorithms for solving VSP were proposed. The authors used the Data Cube framework, in which the search space is represented by bit strings of length equal to the number of nodes in the cube. Each bit indicates whether the corresponding candidate is selected. One of the simplest algorithms considered in [32] is *Random Sampling* which looks as follows: a) randomly sample a bit string b) check whether all constraints are satisfied, and c) estimate the benefit. The solution that satisfies the constraints and has the highest benefit is returned as the answer. The algorithm can be a solution of choice in scenarios when the computation budget is very limited [15].

5.3.2 Local Search. In the same paper the authors proposed more involved *Iterative Improvement* (abbreviated as II) and *Simulated Annealing* (SA) algorithms. To explain the main ideas behind them we need to define the notion of neighborhood of two solutions. Let us introduce three types of actions: 1) select new candidate, 2) replace one candidate with another one, 3) remove candidate. Two solutions are called neighbors if one of them can be obtained from another by a single action. II algorithm implements random transitions only to neighbors C' with a higher benefit $\mathfrak{B}(C')$. In SA algorithm, a (random) transition to neighbors with a lower benefit is possible, but the probability of such a transition is the less the smaller the benefit of the neighbor is. The intuition behind this is as follows: local optima may well be connected by a short path via candidates with a smaller benefit, but frequent transitions to less useful solutions make the search longer. As II algorithm makes transitions only to useful neighbors, it converges to good solutions rather quickly, although it may get stuck in isolated regions. In turn, the quality of solutions obtained by SA is higher. To combine the advantages of both algorithms, a *Two-Phase Optimization procedure (2PO)* is employed, which was previously proposed in [29]. By using II algorithm, 2PO first converges to an area of potential interest and then explores it a more detail by SA algorithm.

In [12], an algorithm based on Simulated Annealing is proposed. In this work, solutions are considered to be neighbors if their representation strings differ only at one position. The algorithm is rather simply adapted to a parallel computing scenario [13], in which communication between processes is not required when moving to neighbors: several SA procedures are run independently and the best solution they find is returned as the final answer.

5.3.3 Genetic Algorithm. The genetic algorithm is a well-known approach to solve NP-hard problems; it is based on a search procedure inspired by the principles of natural selection and genetics. The search is carried out by an iterative improvement of generations, each of which is represented by a population. From each generation, with the help of *mutations* and *crossovers*, a new generation is created, from which the best individuals are *selected*. The principle step here is to present candidates as a population. To both a) reflect the existence of multiple individual query plans and b) represent the entire set of candidates, Horng et al. [26] propose the following schema. By using a special *query-plan string* (qps), they record which execution plans for queries are selected, and concatenate it with a *view string* (vs), which represents selected candidates. Mutation is implemented by changing the value in a random position in the string, while the crossover is implemented as a *cut-and-swap* operation separately on *vs* and *qps*. To keep the population size limited, a random number of the best candidates is kept according to their benefits. Also, at each iteration, the candidates are improved via local search, i.e., in fact, the proposed procedure refers to the class of *Genetic Local search algorithms* [34].

In the above presented schema, as in most evolutionary algorithms, candidates are compared based on their benefit. But if the optimization problem includes a space constraint then it must also be taken into account. Indeed, if one candidate has a slightly less benefit than another one, but implies a significantly less expense, then it might be worth selecting it. In [36], the authors suggested to use the technique of penalty functions, which allows for a) choosing cheaper candidates from those having the same benefit and b) skip solutions that violate constraints. This approach helps to *reduce the length* of the shortest path to the optimum and to *avoid complex search landscapes* where valid regions are separated by invalid ones.

An alternative approach is randomized *stochastic ranking*, in which candidates are compared with probability p by their benefit, and with probability $1 - p$ by the size of the remaining expense budget. In [62] Yu et al. proposed an evolutionary algorithm which generalizes the selection step. Population selection is implemented similar to the bubble sort, where stochastic comparison is used instead of the standard comparison. Sorting ends if no permutations occurred at the iteration. After this, the first k individuals are selected as the result.

[9] this is a standard technique for reducing the size of processed data

[10] this holds, because the framework considered does not reflect alternative computation paths, so there is no way to avoid computing a child when executing an ancestor

5.4 Hybrid algorithms

One more important step in the development of selection algorithms was made by hybrid approaches which compensate for weak points of some methods with the advantages of the others.

5.4.1 Early Approaches. One of the first ideas in this direction is proposed in [65]. The authors consider a setting in which a query has several plans, and their algorithm has two stages. The first stage is to select a plan for every query and build an expression forest $\mathcal{F}$ over them. The second stage is to select candidates from $\mathcal{F}$. At each of these stages, two basic algorithms are applied: a greedy [20, 61] and an evolutionary [26] one. The authors tried 4 combinations of algorithms in total. In experiments they came to the conclusion that going beyond individually optimal plans for queries can greatly improve the quality of obtained solutions. Also, the most accurate solution was obtained when the problem at level 2 was solved by an evolutionary algorithm. This confirms the hypothesis that *correctly estimated benefits are essential for making good selection*, because evolutionary algorithms do not rely on greedy heuristics and provides more accurate benefit estimates.

In [68], the authors combine the ideas of greedy and randomized approaches for efficient selection of views together with indexes. At the first step, similarly to the techniques described in Section 5.2.3, they expand the candidate space by finding common subexpressions and compensations (see Example 4.2). Benefits are calculated by using the optimizer's *'what-if' mode* (see Section 4.2). Based on the obtained values, a greedy algorithm is applied which has the property that in case an index is selected at some iteration, the algorithm tries to select the corresponding view simultaneously (provided there is enough space). Then an iterative improvement is performed for the solution obtained by the greedy algorithm, which randomly picks several unselected candidates instead of selected ones. This allows for compensating the non-optimality of the greedy algorithm.

5.4.2 Modern Approaches. In [31], the View Selection Problem is considered in a practical DBaaS scenario. The authors note that % of common computations between queries from real workloads is typically quite large, while savings from their reuse can be as high as 40% (of the available computing resources). By using the recurrent nature of queries in their scenario, the authors manage to overcome two problems at once. First, to deal with inaccurate cost estimates from the optimizer the authors propose to use statistics on previously executed queries. This is known in the literature as *feedback loop* technique. Secondly, they implement view selection in an online scenario (i.e., when the workload is not known in advance) by using information about the workload observed in the past and by assuming that queries in the future will be similar. To quickly search for relevant statistics and check for useful materialized views, they employ so-called *signature technique* that takes into account the *recurrent* nature of queries and is used as an efficient search filter. The authors also take into account that queries are executed in parallel and therefore, decisions on the materialization of views are also made in parallel. As a consequence, the same view can be created multiple times, which wastes both memory and time. To deal with this, the authors propose a special *early materialization* technique. In short, the idea is to: a) not to materialize the same computation multiple times, b) make the materialized computation available for reuse until the complete execution of the query that triggered its materialization. Even if the triggering query is rolled back, the materialized view remains available for reuse.

In [30], a number of novel ideas on the selection procedure was proposed by the authors in **BigSub** solution. In their approach, selection is initially formulated as a ILP problem. However, in order to model the non-linear behavior of functions, one has to introduce a large number of variables, which makes the resulting ILP problem infeasible. The main idea of BigSub is to use an iterative algorithm which allows for splitting the original problem into a big number of independent subtasks. The decision which candidates should be selected at each iteration is made by a *random flip* function. After a set of candidates is fixed the optimal ways to use them are decided *independently* for each of the queries by using solutions to ILP subtasks. Based on the estimated benefit for selected candidates the distribution of the $flip$ function is changed and then the next iteration is performed. To optimize the solution of ILP subtasks, a number of heuristics are proposed for reducing the search space and changing strategies depending on the workload. By using the idea of problem decomposition and by employing a vertex-centric graph processing model (like Giraph [39] or GraphLab in [37]), the authors managed to cope with the size of the original problem, which was beyond the capacity of the reference ILP solver Gurobi [23].

The state-of-the-art solution to VSP is given by **RLView** algorithm from [63] which extends the BigSub approach by replacing the procedures of random flipping and benefit estimation with ML algorithms. This work is not limited to the online scenario and assumes that there is enough budget to factorize workload subexpressions into semantic equivalence classes (e.g., by using *Equitas* equivalence checker [67]), which provide a fairly rich candidate space. Then in each equivalence class, the cheapest representative is chosen, which significantly reduces the set of candidates. The authors approach the problem of inaccurate predictions of the optimizer by using a *Wide-Deep model*, which is a neural network capable of modeling both linear and highly non-linear dependencies of the execution time on input parameters [6]. The model is trained on collected statistics, which is similar to the feedback loop technique [42, 53]. Various ways of encoding strings, keywords, and table schemas in queries are employed, over which recurrent LSTM networks [17] are run to capture the overall structure of the query. The constraints in the selection problem are taken into account in the form of a regularizer. The regularization coefficient is defined by converting everything into a universal measure unit (money).

The intuition behind the RLView approach can be explained as follows. In fact, "*BigSub has no memorization ability and it does not converge to a global optimal solution, because there is no information sharing between different iterations*". Therefore, inspired by the success of Reinforcement Learning [55], the authors essentially proposed a randomized (*but informed*) search procedure based on a Markov Decision Process (MDP). Typically, a MDP is defined by 4 parameters: a state space S, an action space A, a distribution $P_a(s, s')$ of the probability of transition between states s and s' by an action a, and a distribution $R_a(s, s')$ of reward r when moving from state s to s' by a. The task is to find a strategy $\pi : S \times A \rightarrow \{0, 1\}$

which gives, on average, the highest total reward. The fundamental difference from BigSub is that this model of the flip function a) takes into account the current state of s and b) learns over time. In RLView algorithm, a state s is a set of selected candidates and a description of how to use them, an action a means change in the decision on the selection of a candidate, and a reward r reflects the change in the total benefit due to an action. The optimal view selection strategy is searched by using Q-Learning [58]. In particular, the authors build a neural network *DQN* [45, 57], which is trained to predict the potential benefit from each of the actions in a given state. In experiments, the authors established 3 SotA results on different datasets and also confirmed the hypothesis that *the accuracy of estimated benefits plays one of the key roles* in solving the selection problem. Both RLView and BigSub demonstrated best results when using a predictive model instead of optimizer estimates.

5.5 Summary on Algorithms

As we have shown in this section, the algorithms for solving the selection problem can be roughly divided into three classes. Heuristic algorithms, based on intuitive assumptions for estimating the benefit of candidates and reducing the candidate space, aim at providing fast approximate solutions. However, the efficiency of these algorithms strongly depends on the size of the input data, which makes them problematic for high dimensions. Randomized algorithms try to overcome this limitation by offering a trade-off between the accuracy and speed of computations, but they provide no guarantees on the quality of obtained solutions. Recent advances in solving the Selection Problem have been made with hybrid solutions that utilize the strengths of approaches from different domains, including Machine Learning.

We conclude that there is a number of similar points in these algorithms and we summarize the most important of them below.

The choice of candidate space. The quality of obtained solutions strongly depends on the space of candidates. To obtain a good candidate space one can use plan building strategies, plan merging techniques, and methods for finding common subexpressions (Sections 5.2.3 and 5.4.1). It is also possible to use space reduction methods, which provide more flexibility in building an efficient selection algorithm (Section 5.2.2).

Tree structure of candidates. In selection problems, candidates are typically tree nodes while benefits are given as weights of lightest computation paths. Relationships between candidates and the way benefits / expenses are computed can greatly reduce the search space and speed up algorithms (Section 5.2.2).

Dependence on accurate cost estimates. Several implementations have confirmed the intuitively clear fact that *the quality of solutions obtained by selection algorithms strongly depends on the accuracy of benefit estimates for candidates* (Sections 5.4.1 and 5.4.2). Recent advances in solving the View Selection Problem are due to approaches *which do not rely on DB optimizer*. Modern algorithms employ history-based (feedback loop) techniques and ML based prediction models.

Modularity of solutions and the use of ML. There is a general trend of developing modular solutions which compromise between advantages and disadvantages of several types of algorithms (Sections 5.3.2 and 5.4.2). Since the dependencies between objects involved in the Selection Problem are highly complex, it is hard to model them explicitly. However, a large amount of data describing these dependencies is available. Given the fact that ML is an excellent tool for discovering complex patterns in data, it is natural to believe that *ML will be used extensively in selection algorithms*.

6 CHALLENGES AND OPEN PROBLEMS

6.1 Design of Candidates

Candidate Space. Whatever the selection algorithm is, to find a good solution, it must be at least be contained in the chosen candidate space. The question of choosing the right space is highly non-trivial. It has been shown in [7] that an optimal solution can not be found if one builds a candidate space upon subexpressions of queries. On the other hand, it is impractical to work with large subexpression spaces. Hence, it is necessary to study ways how to *constrain the candidate space* while providing guarantees on the quality of contained solutions. Heuristic approaches have been already attempted for this purpose (Section 5.2.2), but we believe that there is much room for new results and advanced techniques in this area.

Computational Complexity. The formal complexity analysis has made a significant contribution to the development of selection algorithms (Section 4.3). Based on the state-of-the-art results we believe that it is worthwhile to study the approximability of the Selection Problem for AND-OR-DAGs. Understanding this issue will facilitate building efficient selection algorithms for this expressive representation framework. The computational complexity is also worth investigating for AND-DAG and Data Cube frameworks, since for these representations the (non-)existence of an exact polynomial solution [22, 33] has not been yet established. There are also hopes for positive computational results on the little explored classes of binary AND-OR-DAGs.

6.2 Benefit estimation

Model architecture. Correctly estimating the benefit of candidates is an important part in solving the Selection Problem (Section 5.4). One way to obtain accurate estimates is to build a special benefit prediction model. We noted that using the tree structure of queries allows for designing more optimal selection algorithms (Section 5.2), and we believe this knowledge should be employed in prediction models as well. While searching for new architectures, it is also worth adopting similar solutions from other fields, among which we highlight Tree Convolution networks [56] and TreeLSTM modification of recurrent neural networks [56].

Encoding. To built a vectorized representation of a query, it is important to correctly *encode* its keywords and (sub)expressions. In [42], a special encoding technique similar to word2vec [43] was proposed which allows for encoding more information and significantly improves prediction accuracy. State-of-the-art techniques for encoding plans in prediction models typically use one-hot encoding and linear transformations, which leaves space for advanced encoding techniques.

6.3 Diverse Scenarios

Dynamic. The algorithms considered in this paper use an implicit assumption that candidates persist until the entire given workload

is executed. But if a candidate is found to be useful for executing only a part of the workload, it can be replaced at some point [19, 64]. Controlling the order of query execution and dynamically changing the set of selected candidates can significantly increase the overall performance. We think that this feature should become a part of modern selection algorithms.

Distributed. Due to the trend towards scalable computing systems, in order to develop efficient solutions to the Selection Problem, it is necessary to take into account data location in benefit modeling, similar to the ideas from [5].

Partial Selection. In some situations one can partially save a candidate, for example, to keep only its most frequently requested part. Then, in processing a query, most of the data can be retrieved by reusing the candidate, while the rest of the data can be computed from base relations [38]. A similar approach was implemented by using *hotspot* tables in paper [66]. We think that these techniques should be further developed, since they allow for reducing the size of candidates to store, thus avoiding expensive disk reads.

6.4 Unified Selection Framework

Interconnections of Problems. The common nature of selection tasks in different domains suggests reuse of ideas. We see that the idea of storing frequent objects from classical caching is adopted in View Selection [10] and Index Selection [51]. Reuse of techniques can also be observed in the combined index and view selection algorithms (Sections 5.4.1, 5.2.1). We believe there is a strong potential in such approaches. For example, we noticed the non-linear cost of saving in plan caching (see Example 3.6), which leads to the idea to more efficiently store plans and reuse techniques for the maintenance constraint from the field of View Selection.

Evaluation Platform. With the potential of reuse of techniques between different approaches, we see a great benefit in creating a universal platform for evaluating pros and cons of different selection algorithms. In this paper, we are taking the first step in this direction by proposing a general Candidate Selection framework. We face however a number of technical challenges on this way, including platform-dependence and closed code of some implementations, as well as the lack of custom benchmarks.

7 CONCLUSION

In this paper, we highlighted the cross-domain nature of selection algorithms and proposed the framework of Candidate Selection for re-use of these algorithms. The framework covers many aspects of Multi-Query Optimization, ranging from View Selection and Index Selection to Query / Plan Caching. We provided a deep analysis of selection problems in different domains and techniques to address them. We offered a modern classification of selection algorithms, we formulated open issues and challenges and suggested promising directions for future research.

REFERENCES

[1] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R Narasayya. 2000. Automated selection of materialized views and indexes in SQL databases. In *VLDB*, Vol. 2000. 496–505.

[2] Hossein Azgomi and Mohammad Karim Sohrabi. 2018. A game theory based framework for materialized view selection in data warehouses. *Engineering Applications of Artificial Intelligence* 71 (2018), 125–137.

[3] Xavier Baril and Zohra Bellahsene. 2003. Selection of materialized views: A cost-based approach. In *Advanced Information Systems Engineering: 15th International Conference, CAiSE 2003 Klagenfurt/Velden, Austria, June 16–20, 2003 Proceedings 15*. Springer, 665–680.

[4] Surajit Chaudhuri, Mayur Datar, and Vivek Narasayya. 2004. Index selection for databases: A hardness study and a principled heuristic solution. *IEEE transactions on knowledge and data engineering* 16, 11 (2004), 1313–1323.

[5] Leonardo Weiss F Chaves, Erik Buchmann, Fabian Hueske, and Klemens Böhm. 2009. Towards materialized view selection for distributed databases. In *Proceedings of the 12th international conference on extending database technology: advances in database technology*. 1088–1099.

[6] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. 2016. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*. 7–10.

[7] Rada Chirkova, Alon Y Halevy, and Dan Suciu. 2002. A formal perspective on the view selection problem. *The VLDB Journal* 11 (2002), 216–237.

[8] Rada Chirkova, Jun Yang, et al. 2012. Materialized views. *Foundations and Trends® in Databases* 4, 4 (2012), 295–405.

[9] Bobbie Cochrane, Hamid Pirahesh, and Markos Zaharioudakis. [n.d.]. APPLYING MASS QUERY OPTIMIZATION TO SPEED UP. ([n. d.]).

[10] Shaul Dar, Michael J Franklin, Bjorn T Jonsson, Divesh Srivastava, Michael Tan, et al. 1996. Semantic data caching and replacement. In *VLDB*, Vol. 96. 330–341.

[11] Kalen Delaney. 2007. *Query tuning and optimization*. Microsoft Press, Redmond, WA.

[12] Roozbeh Derakhshan, Frank KHA Dehne, Othmar Korn, and Bela Stantic. 2006. Simulated Annealing for Materialized View Selection in Data Warehousing Environment.. In *Databases and applications*. 89–94.

[13] Roozbeh Derakhshan, Bela Stantic, Othmar Korn, and Frank Dehne. 2008. Parallel simulated annealing for materialized view selection in data warehousing environments. *Lecture Notes in Computer Science* 5022 (2008), 121–132.

[14] AA Diwan, S Sudarshan, and Dilys Thomas. 2006. Scheduling and caching in multi-query optimization. In *International Conference on Management of Data COMAD, Delhi, India*.

[15] César A. Galindo-Legaria, Arjan Pellenkoft, and Martin L. Kersten. 1994. Fast, Randomized Join-Order Selection - Why Use Transformations?. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 85–95.

[16] Antara Ghosh, Jignashu Parikh, Vibhuti S Sengar, and Jayant R Haritsa. 2002. Plan selection based on query clustering. In *VLDB'02: Proceedings of the 28th international conference on very large databases*. Elsevier, 179–190.

[17] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. 2016. LSTM: A search space odyssey. *IEEE transactions on neural networks and learning systems* 28, 10 (2016), 2222–2232.

[18] Ashish Gupta, Inderpal Singh Mumick, et al. 1995. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.* 18, 2 (1995), 3–18.

[19] Amit Gupta, S Sudarshan, and Sundar Vishwanathan. 2001. Query scheduling in multi query optimization. In *Proceedings 2001 International Database Engineering and Applications Symposium*. IEEE, 11–19.

[20] Himanshu Gupta et al. 1997. Selection of views to materialize in a data warehouse. In *ICDT*, Vol. 97. 98–112.

[21] Himanshu Gupta and Inderpal Singh Mumick. 1999. Selection of views to materialize under a maintenance cost constraint. In *Database Theory—ICDT'99: 7th International Conference Jerusalem, Israel, January 10–12, 1999 Proceedings*. Springer, 453–470.

[22] Himanshu Gupta and Inderpal Singh Mumick. 2005. Selection of views to materialize in a data warehouse. *IEEE Transactions on Knowledge and Data Engineering* 17, 1 (2005), 24–43.

[23] Gurobi Optimization, LLC. 2023. Gurobi Optimizer Reference Manual. https://www.gurobi.com

[24] Alon Y Halevy. 2001. Answering queries using views: A survey. *The VLDB Journal* 10 (2001), 270–294.

[25] Venky Harinarayan, Anand Rajaraman, and Jeffrey D Ullman. 1996. Implementing data cubes efficiently. *Acm Sigmod Record* 25, 2 (1996), 205–216.

[26] J-T Horng, Y-J Chang, and B-J Liu. 2003. Applying evolutionary algorithms to materialized view selection in a data warehouse. *Soft Computing* 7 (2003), 574–581.

[27] Arvind Hulgeri and S Sudarshan. 2002. Parametric query optimization for linear and piecewise linear cost functions. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 167–178.

[28] Arvind Hulgeri and S Sudarshan. 2003. AniPQO: Almost non-intrusive parametric query optimization for nonlinear cost functions. In *Proceedings 2003 VLDB Conference*. Elsevier, 766–777.

[29] Yannis E Ioannidis and Younkyung Kang. 1990. Randomized algorithms for optimizing large join queries. *ACM Sigmod Record* 19, 2 (1990), 312–321.

[30] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. 2018. Selecting subexpressions to materialize at datacenter scale. *Proceedings of the VLDB*

Endowment 11, 7 (2018), 800–812.

[31] Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc Friedman, Yifung Lin, Konstantinos Karanasos, and Sriram Rao. 2018. Computation reuse in analytics job service at microsoft. In *Proceedings of the 2018 International Conference on Management of Data*. 191–203.

[32] Panos Kalnis, Nikos Mamoulis, and Dimitris Papadias. 2002. View selection using randomized search. *Data & Knowledge Engineering* 42, 1 (2002), 89–111.

[33] Howard Karloff and Milena Mihail. 1999. On the complexity of the view-selection problem. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 167–173.

[34] Antoon Kolen and Erwin Pesch. 1994. Genetic local search in combinatorial optimization. *Discrete Applied Mathematics* 48, 3 (1994), 273–284.

[35] Wilburt Juan Labio, Dallan Quass, and Brad Adelberg. 1997. Physical database design for data warehouses. In *Proceedings 13th International Conference on Data Engineering*. IEEE, 277–288.

[36] Minsoo Lee and Joachim Hammer. 2001. Speeding up materialized view selection in data warehouses using a randomized algorithm. *International Journal of Cooperative Information Systems* 10, 03 (2001), 327–353.

[37] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. 2014. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041* (2014).

[38] Gang Luo. 2006. Partial materialized views. In *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 756–765.

[39] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.

[40] Imene Mami and Zohra Bellahsene. 2012. A survey of view selection methods. *Acm Sigmod Record* 41, 1 (2012), 20–29.

[41] Imene Mami, Remi Coletta, and Zohra Bellahsene. 2011. Modeling view selection as a constraint satisfaction problem. In *Database and Expert Systems Applications: 22nd International Conference, DEXA 2011, Toulouse, France, August 29-September 2, 2011, Proceedings, Part II 22*. Springer, 396–410.

[42] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A learned query optimizer. *arXiv preprint arXiv:1904.03711* (2019).

[43] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[44] Hoshi Mistry, Prasan Roy, S Sudarshan, and Krithi Ramamritham. 2001. Materialized view selection and maintenance using multi-query optimization. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*. 307–318.

[45] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[46] Nils J Nilsson. 1982. *Principles of artificial intelligence*. Springer Science & Business Media.

[47] Kenneth A Ross, Divesh Srivastava, and S Sudarshan. 1996. Materialized view maintenance and integrity constraint checking: Trading space for time. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*. 447–458.

[48] Prasan Roy, S Seshadri, S Sudarshan, and Siddhesh Bhobe. 1998. *Practical Algorithms for multi query Optimization*. Technical Report. Technical report, Indian Institute of Technology, Bombay.

[49] Prasan Roy, Srinivasan Seshadri, S Sudarshan, and Siddhesh Bhobe. 2000. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 249–260.

[50] Thomas P. Runarsson and Xin Yao. 2000. Stochastic ranking for constrained evolutionary optimization. *IEEE Transactions on evolutionary computation* 4, 3 (2000), 284–294.

[51] Praveen Seshadri and Arun Swami. 1995. Generalized partial indexes. In *Proceedings of the Eleventh International Conference on Data Engineering*. IEEE, 420–427.

[52] Mohammad Karim Sohrabi and Vahid Ghods. 2016. Materialized View Selection for a Data Warehouse Using Frequent Itemset Mining. *J. Comput.* 11, 2 (2016), 140–148.

[53] Michael Stillger, Guy M Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO-DB2's learning optimizer. In *VLDB*, Vol. 1. 19–28.

[54] Julia Stoyanovich, Kenneth A Ross, Jun Rao, Wei Fan, Volker Markl, and Guy Lohman. 2008. ReoptSMART: A Learning Query Plan Cache. (2008).

[55] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

[56] Kai Sheng Tai, Richard Socher, and Christopher D Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075* (2015).

[57] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. 2016. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*. PMLR, 1995–2003.

[58] Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8 (1992), 279–292.

[59] Jennifer Widom. 1995. Research problems in data warehousing. In *Proceedings of the fourth international conference on Information and knowledge management*. 25–30.

[60] Jian Yang, Kamalakar Karlapalem, and Qing Li. 1997. Algorithms for materialized view design in data warehousing environment. In *VLDB*, Vol. 97. 136–145.

[61] Jian Yang, Kamalakar Karlapalem, and Qing Li. 1997. A framework for designing materialized views in data warehousing environment. In *Proceedings of 17th International Conference on Distributed Computing Systems*. IEEE, 458–465.

[62] Jeffrey Xu Yu, Xin Yao, Chi-Hon Choi, and Gang Gou. 2003. Materialized view selection as constrained evolutionary optimization. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 33, 4 (2003), 458–467.

[63] Haitao Yuan, Guoliang Li, Ling Feng, Ji Sun, and Yue Han. 2020. Automatic view generation with deep learning and reinforcement learning. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1501–1512.

[64] Chuan Zhang, Jian Yang, and Kamalakar Karlapalem. 2003. Dynamic materialized view selection in data warehouse environment. *Informatica (Slovenia)* 27, 4 (2003), 451–460.

[65] Chuan Zhang, Xin Yao, and Jian Yang. 2001. An evolutionary approach to materialized views selection in a data warehouse environment. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 31, 3 (2001), 282–294.

[66] Jingren Zhou, Per-Ake Larson, Jonathan Goldstein, and Luping Ding. 2006. Dynamic materialized views. In *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 526–535.

[67] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Dong Xu. 2019. Automated verification of query equivalence using satisfiability modulo theories. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1276–1288.

[68] Daniel C Zilio, Calisto Zuzarte, Sam Lightstone, Wenbin Ma, Guy M Lohman, Roberta J Cochrane, Hamid Pirahesh, Latha Colby, Jarek Gryz, Eric Alton, et al. 2004. Recommending materialized views and indexes with the IBM DB2 design advisor. In *International Conference on Autonomic Computing, 2004. Proceedings.* IEEE, 180–187.