Fast and Effective Distribution-Key Recommendation for Amazon Redshift

Panos Parchas
Amazon Web Services
Berlin, Germany
parchp@amazon.com

Yonatan Naamad
Amazon Devices
Sunnyvale, CA, USA
ynaamad@amazon.com

Peter Van Bouwel
Amazon Web Services
Dublin, Ireland
pbbouwel@amazon.com

Christos Faloutsos
Carnegie Mellon University & Amazon
faloutso@amazon.com

Michalis Petropoulos
Amazon Web Services
Palo Alto, CA, USA
mpetropo@amazon.com

ABSTRACT

How should we split data among the nodes of a distributed data warehouse in order to boost performance for a forecasted workload? In this paper, we study the effect of different data partitioning schemes on the overall network cost of pairwise joins. We describe a generally-applicable data distribution framework initially designed for Amazon Redshift, a fully-managed petabyte-scale data warehouse in the cloud. To formalize the problem, we first introduce the *Join Multi-Graph*, a concise graph-theoretic representation of the workload history of a cluster. We then formulate the *"Distribution-Key Recommendation"* problem – a novel combinatorial problem on the *Join Multi-Graph*– and relate it to problems studied in other subfields of computer science. Our theoretical analysis proves that *"Distribution-Key Recommendation"* is **NP**-complete and is hard to approximate efficiently. Thus, we propose BAW, a hybrid approach that combines heuristic and exact algorithms to find a good data distribution scheme. Our extensive experimental evaluation on real and synthetic data showcases the efficacy of our method into recommending optimal (or close to optimal) distribution keys, which improve the cluster performance by reducing network cost up to **32x** in some real workloads.

PVLDB Reference Format:
Panos Parchas, Yonatan Naamad, Peter Van Bouwel, Christos Faloutsos, Michalis Petropoulos. Fast and Effective Distribution-Key Recommendation for Amazon Redshift. *PVLDB*, 13(11): 2411-2423, 2020.
DOI: https://doi.org/10.14778/3407790.3407834

1. INTRODUCTION

Given a database query workload with joins on several relational tables, which are partitioned over a number of machines, how should we distribute the data to reduce network cost and ensure fast query execution?

Amazon Redshift [12, 2, 3] is a massively parallel processing *MPP* database system, meaning that both the storage and processing of a cluster is distributed among several machines (*compute nodes*). In such systems, data is typically distributed row-wise, so that each row appears (in its entirety) at one compute node, but distinct rows from the same table may reside on different machines. Like most commercial data warehouse systems, Redshift supports 3 distinct ways of distributing the rows of each table:

- *"Even"* (= uniform/random), which distributes the rows among compute nodes in a round-robin fashion;
- *"All"* (= replicated), which makes full copies of a database table on each of the machines;
- *"Dist-Key"* (=fully distributed = hashed), which hashes the rows of a table on the values of a specific attribute known as *distribution key* (DK).

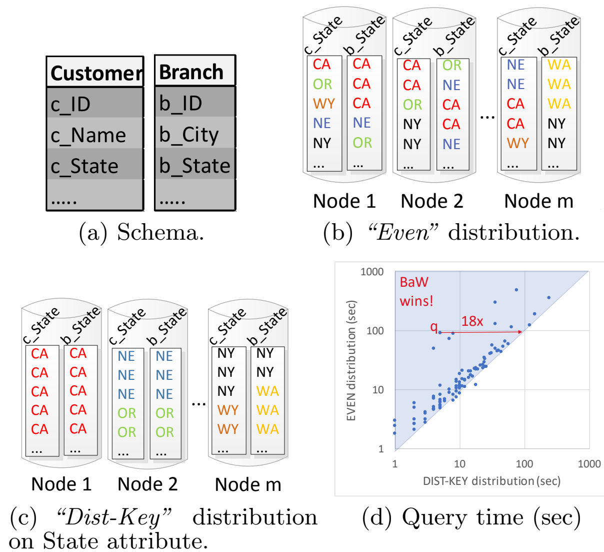
In this work, we focus on the latter approach. Our primary observation is that, if both tables participating in a join are distributed on the joining attributes, then that join enjoys great performance benefits due to minimized network communication cost. In this case, we say that the two tables are *collocated* with respect to the given join. The goal of this paper is to minimize the network cost of a given query workload by carefully deciding which (if any) attribute should be used to hash-distribute each database table.

Figure 1 illustrates collocation through an example. Consider the schema of Figure 1(a) and the following query Q1:

```
-- query Q1
SELECT *
FROM Customer JOIN Branch
    ON c_State = b_State;
```

With the records distributed randomly (*"Even"* distribution style) as in Figure 1(b), evaluating Q1 incurs very high network communication cost. In particular, just joining the red "CA" rows of '`Customer`' from Node 1 with the corresponding "CA" rows of '`Branch`' from Node 2 requires that at least one of the nodes transmits its contents to the other, or that both nodes transmit their rows to a third-party node. The same requirement holds for all states and for all pairs of nodes so that, ultimately, a large fraction of the database must be communicated in order to compute this single join.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407834



(a) Schema. (b) *"Even"* distribution.

(c) *"Dist-Key"* distribution on State attribute. (d) Query time (sec)

Figure 1: BaW wins. (a-c) Data distribution is important: visually, *"Dist-Key"* needs less communication, for the Join on `State`, when tables are spread over m nodes. (d) Our method BaW makes almost all TPC-DS queries faster, up to 18x (execution times of *"Even"* vs proposed BaW in seconds).

This is in stark contrast with Figure 1(c), in which all "CA" rows are on Node 1, all "NE" rows are on Node 2, and so on. This is the best-case scenario for this join, in that no inter-server communication is needed; all relevant pairs of rows are already *collocated* and the join can be performed locally at each compute node. This ideal setup happens whenever the distribution method *"Dist-Key"* is chosen for both tables, and the distribution key attributes are chosen to be `Customer.c_State` and `Branch.b_State` respectively, hashing all records with matching state entries to the same compute node. This yields significant savings in communication cost, and in overall execution time.

1.1 Informal problem definition

The insight that collocation can lead to great savings in communication costs raises a critical question: How can we achieve optimal data distribution in the face of multiple tables each participating in multiple joins, which may or may not impose conflicting requirements? We refer to this as the *"Distribution-Key Recommendation"* problem. Our solution, Best of All Worlds (BaW), picks a distribution key for a subset of the tables so to maximally collocate the most impactful joins.

Figure 1 (d) illustrates the query performance of the TPC-DS [19] queries (filled circles, in the scatter-plot) loaded on a Redshift cluster. The x-axis (resp. y-axis) corresponds to the query execution time in seconds for the BaW (resp. *"Even"*) distribution. BaW consistently performs better, and rarely ties: most points/queries are above the diagonal, with a few on or below the diagonal[1]. For some queries, like q, the savings in performance reach 18x (axes on log scale).

[1]The lone datapoint significantly below the diagonal is further discussed in Section 6.2.

Despite the importance of the problem, there have been few related papers from the database community. The majority of related methods either depend on a large number of assumptions that are not generally satisfied by most commercial data warehouse systems, including Amazon Redshift, or provide heuristic solutions to similar problems. To the best of our knowledge, this work is the first to propose an efficient, assumption-free, purely combinatorial, provably optimal data distribution framework based on the query workload. Furthermore, our method, BaW, is applicable to any column store, distributed, data warehouse. It is standalone and does not require any change in the system's architecture.

The contributions of the paper are:

- **Problem formulation**: We formulate the *"Distribution-Key Recommendation"* (DKR) problem, as a novel graph theoretic problem and we show its connections to graph matching and other combinatorial problems. Also, we introduce the *Join Multi-Graph*, a concise and light-weight graph-theoretic representation of the join characteristics of a cluster.
- **Theoretical Analysis**: We analyze the complexity of DKR, and we show it is **NP**-complete.
- **Fast Algorithm**: We propose BaW, an efficient meta-algorithm to solve DKR; BaW is extensible, and can accommodate any and every optimization sub-method, past or future.
- **Validation on real data**: We experimentally demonstrate that the distribution keys recommended by our method improve the performance of real Redshift clusters by up to **32x** in some queries.

The rest of the paper is organized as follows. Section 2 surveys the related work. Section 3 introduces the *Join Multi-Graph* and mathematically formulates the *"Distribution-Key Recommendation"* problem (DKR). Section 4, presents a theoretical analysis of DKR and proves its complexity. Section 5 proposes BaW, an efficient graph algorithm to solve DKR. Section 6 contains an extensive experimental evaluation of our method on real and synthetic datasets, on Amazon Redshift. Lastly, Section 7 concludes the paper.

2. BACKGROUND AND RELATED WORK

This section starts with an overview of previous papers on workload-based data partitioning. Then, it describes related graph problems motivating our BaW algorithm.

2.1 Data partitioning

Workload-based data partitioning has been well studied from various perspectives, including generation of indices and materialized views [1], partitioning for OLTP workloads [5, 21, 23], and others. However, our main focus here is data partitioning in OLAP systems such as Amazon Redshift. There have been two main directions of research depending on the interaction with the query optimizer: 1) *optimizer modifying* techniques, which require alteration of the optimizer and 2) *optimizer independent* techniques, which work orthogonally to the optimizer.

2.1.1 Optimizer modifying

The first category heavily alters the optimizer, exploiting its cost model and its internal data structures to suggest a good partitioning strategy. Nehme and Bruno [20] extend

the notion of MEMO[2] of traditional optimizers to that of *workload MEMO*, which can be described as the union of the individual MEMOs of all queries, given the workload and the optimizer's statistics. The cost of each possible partitioning of the data is approximated, in a way similar to the cost estimation of potential query execution plans in traditional optimizers. This technique ensures optimal data partitioning based on the given workload. Unfortunately, it does not have general applicability as it requires severe alteration of the query optimizer. In addition, the generation of workload MEMO is a rather expensive process as it requires several minutes for a workload of only 30 queries. In order to give accurate recommendations, our method utilizes a pool of tens of thousands of queries in just few seconds.

Other methods utilize the *What-if Engine*, which is already built into some query optimizers [22, 11]. The main idea is to create a list of candidate partitions C for each table and evaluate the entire workload on various combinations of C using statistics. Due to the exponential number of such combinations, [22] utilizes Genetic Algorithms to balance the elements of directed and stochastic search. Unfortunately, although faster than deep integration, shallow integration has several disadvantages: first, the search space of all feasible partitioning configurations is likely to become extremely large, due to the combinatorial explosion of combinations. Second, although smart techniques limit the search space, the approach is still very expensive because each query in the workload needs to be evaluated in several candidate *what-if* modes. Lastly, many systems (including Amazon Redshift) do not support a *what-if* mode in the optimizer, which limits the applicability of the approach in large commercial systems.

2.1.2 Optimizer independent

The optimizer independent methods are closer to our work [30, 10, 26, 24, 7, 28].

Zilio et. al. [30] also use a weighted graph to represent the query workload, similarly to our approach. However, the graph weights correspond to the *frequency* of operations and do not capture the *importance* of different joins, as in our case. This could mistakenly favor the collocation of cheaper joins that appear more often in the workload, over expensive joins that are less common. [30] proposes two heuristic algorithms: *IR*, a greedy heuristic approach (similar to the NG baseline of our experiments) that picks the best distribution key for each table in isolation, and *Comb*, an exhaustive search algorithm that is guided by heuristic pruning and depends on optimizer estimates. Similarly, methods of [10, 26] follow a greedy heuristic approach without any optimality guarantee[3]. However, as Theorem 2 in Section 4 shows, no polynomial-time heuristic can give a worst-case approximation guarantee within any constant factor (say 1%) of the optimal solution, unless $P = NP$. Unlike previous approaches, our work describes an optimal solution (see Equation 2 in Section 5.1), provides an in-depth complexity analysis (NP-hard, see Theorem 2 in Section 4) and, in our experiments, the proposed method BaW always reaches the optimal within minutes (see Section 6.2).

[2] A search data structure that stores the alternative execution plans and their expected cost.

[3] The algorithm of [26] is similar to one of our heuristic methods, namely RC (see Section 5.2).

Table 1: Comparison to other methods.

	BaW	**[30]**	**[10]**	**[26]**	**[24], [7], [28]**
1. *provably optimal*	✓	✗	✗	✗	✗
2. *complexity analysis*	✓	✗	✗	✗	✗
3. *deployed*	✓	?	?	✓	?
4. *cost-aware*	✓	✗	✓	✓	✓
5. *extensible algorithm*	✓	✗	✗	?	✗
6. *schema independent*	✓	✓	✓	?	✗
7. *no indices*	✓	✓	✓	✓	✗
8. *no replication*	✓	✓	✓	✗	✗

Stöhr et. al. [24] introduce an algorithm for data allocation in a shared disk system assuming the star schema architecture. They propose hierarchical fragmentation of the fact table and a bitmap index for all combinations of joins between the fact table and the dimensions. Then, they perform a simple round robin distribution of the fact table and the corresponding bitmap indices among the compute nodes. Similarly, [7, 28] propose techniques that require the construction and maintenance of indices, and allow some level of replication. Unfortunately, these approaches impose severe architectural restrictions and are not generally applicable to large MPP systems that do not support indices, such as Amazon Redshift. In addition, they entail extra storage overhead for storing and maintaining the indices. As we show in our experimental evaluation, real workloads follow a much more involved architecture that cannot always be captured by star/snowflake schemata.

Table 1 summarizes the optimizer independent methods, based on the following critical dimensions:

1. *provably optimal*: the proposed algorithm provably converges to the optimal solution,
2. *complexity analysis*: theoretical proof of the hardness of the problem,
3. *deployed*: the approach is deployed at large scale,
4. *cost-aware*: the problem definition includes the cost of joins (and not just their frequency),
5. *extensible algorithm*: the solution is an extensible meta-algorithm that can incorporate *any* heuristic,
6. *schema independent*: no architectural assumptions are made on the input schema (e.g., star/snowflake),
7. *no indices*: no auxiliary indices need to be maintained,
8. *no replication*: space-optimal, no data replication.

A green ✓ (resp: red ✗) in Table 1 indicates that a property is supported (resp: not supported) by the corresponding related paper, whereas a '**?**' shows that the paper does not provide enough information. As Table 1 indicates, none of the above methods matches all properties of our approach.

2.2 Related graph problems

The following classes of graph theoretical problems are used in our theoretical analysis and/or algorithmic design.

2.2.1 Maximum Matching

Let a weighted undirected graph $G = (V, E, w)$. A subgraph $H = (V, E_H, w) \sqsubseteq G$ is a *matching* of G if the degree of each vertex $u \in V$ in H is at most one. The *Maximum Matching* of G is a matching with the maximum sum of

weights. In a simple weighted graph, the maximum weight matching can be found in time $O(|V||E| + |V|^2 \log |V|)$ [9, 4]. The naive greedy algorithm achieves a 1/2 approximation of the optimal matching in time $O(|V| \log |V|)$, similar to a linear-time algorithm of Hougardy [14].

2.2.2 *Maximum Happy Edge Coloring*

In the MAXIMUM HAPPY EDGE COLORING (MHE) problem, we are given an edge-weighted graph $G = (V, E, w)$, a color set $\mathcal{C} = \{1, 2, \cdots, k\}$, and a partial vertex coloring function $\varphi : V' \to \mathcal{C}$ for some $V' \subsetneq V$. The goal is to find a (total) color assignment $\varphi' : V \to \mathcal{C}$ extending φ and maximizing the sum of the weights of mono-colored edges (i.e. those edges (u, v) for which $\varphi'(u) = \varphi'(v)$). This problem is NP-hard, and the reductions in [6], [17], and [29] can be combined to derive a 951/952 hardness of approximation. The problem has recently been shown to admit a 0.8535-approximation algorithm [29].

2.2.3 *Max-Rep / Label Cover*

The MAX-REP problem, known to be equivalent to LABEL-COVER$_{\max}$, is defined as follows [16]: let $G = (V, E)$ be a bipartite graph with partitions A and B each of which is further partitioned into k disjoint subsets $A_1, A_2, \cdots A_k$, and $B_1, B_2, \cdots B_k$. The objective is to select exactly one vertex from each A_i and B_j for $i, j = 1, \cdots, k$ so to maximize the number of edges incident to the selected vertices. Unless NP $\subseteq$ Quasi-P, this problem is hard to approximate within a factor of $2^{\log^{1-\epsilon} n}$ for every $\epsilon > 0$, even in instances where the optimal solution is promised to induce an edge between every A_i, B_j pair containing an edge in G.

3. PROBLEM DEFINITION

Our data partitioning system collocates joins to maximally decrease the total network cost. Thus, we restrict our attention solely to join queries[4].

DEFINITION 1. *Let $Q = \{q_1, q_2, \cdots, q_n\}$ be the query workload of a cluster. For our purposes each query is viewed as a set of pairwise joins, i.e., $q_i = \{j_{i1}, j_{i2}, \cdots, j_{ik}, \cdots j_{im}\}$. Each join is defined by the pair of tables it joins $(t1_k, t2_k)$, the corresponding join attributes $(a1_k, a2_k)$ and the total cost of the join in terms of processed bytes (w_k), i.e., $j_k =< t1_k, t2_k, a1_k, a2_k, w_k >$.*

For instance, in the example provided in Figure 1, if the query $Q1$ appeared 10 times in Q, each yielding a cost w, then the query is represented as

$$Q1 =< Customer, Branch, \texttt{c_State}, \texttt{b_State}, 10w > .$$

DEFINITION 2. *A join $j_k =< t1_k, t2_k, a1_k, a2_k, w_k >$ is* ***collocated****, if tables $t1_k$ and $t2_k$ are distributed on attributes $a1_k$ and $a2_k$ respectively.*

Note that if a join j_k is *collocated*, then at query time we benefit from not having to redistribute w_k bytes of data through the network. The problem we solve in this paper is, given the workload Q, identify which distribution key to choose for each database table in order to achieve the highest benefit from collocated joins. Section 3.1 first introduces *Join Multi-Graph*, a concise representation of the typical workload of a cluster. Section 3.2 formally defines the *"Distribution-Key Recommendation"* problem on the *Join Multi-Graph*.

3.1 Proposed structure: Join Multi-Graph

Given a cluster's query workload Q, consider a weighted, undirected multigraph $G_Q = (V, E, w)$, where V corresponds to the set of tables in the cluster and E contains an edge for each pair of tables that has been joined at least once in Q. Also, let the attribute set $\mathcal{A}_u$ of a vertex $u \in V$ correspond to the set of u's columns that have been used as join attributes at least once in Q (and, thus, are good candidates for DKs). Each edge $e = (u.x, v.y)$ with $\{u, v\} \in V$, $x \in \mathcal{A}_u$ and $y \in \mathcal{A}_v$ encodes the join '*u JOIN v ON* $u.x = v.y$'[5].

The weight $w(e) : E \to \mathbb{N}^+$ represents the cumulative number of bytes processed by that join in Q[6] and quantifies the benefit we would have from collocating the corresponding join[7]. If a join occurs more than once in Q, its weight in G_Q corresponds to the sum of all bytes processed by the various joins[8]. Since two tables may be joined on more than one pair of attributes, the resulting graph may also contain parallel edges between two vertices, making it a multigraph.

Figure 2(a) illustrates our running example of a *Join Multi-Graph* $G_Q = (V, E, w)$. The tables that have participated in at least one join in Q correspond to the vertex set V. An edge represents a join. The join attributes are denoted as edge-labels near the corresponding vertex. The cumulative weight of a join is illustrated through the thickness of the edge; a bold (resp. normal) edge corresponds to weight value 2 (resp. 1). For instance, table B was joined with table D on $B.b = D.d$ and on $B.b_1 = D.d_1$ with weights 1 and 2 respectively. Finally, the attribute set (i.e., the set of join attributes) of table B is $\mathcal{A}_B = \{b, b_1\}$.

G_Q is a concise way to represent the join history of a cluster. Independently of the underlying scheme, the *Join Multi-Graph* contains all valuable information of the join history. It is a very robust and succinct representation that adapts to changes in workload or data: if we have computed G_Q for a query workload Q, we can incrementally construct $G_{Q'}$ for workload $Q' = Q \cup \{q\}$ for any new query q, by increasing the weight of an existing edge (if the join has already occurred in the past) or by adding a new edge (if q contains a new join). Similarly, if the database tables have increased/decreased considerably in size, this mirrors to the weight of the join.

The number of vertices in the graph is limited by the number of tables in the database, which does not usually exceed several thousands. The number of edges is limited by the number of distinct joins in the graph. In practice, not all join attributes are good candidates for distribution keys: columns with very low cardinality would result in high

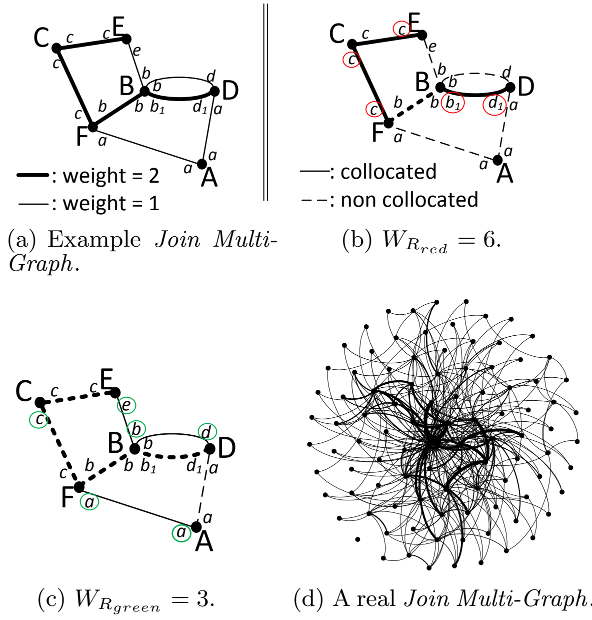
[4]Aggregations could also benefit from collocation in some cases, but the benefit is smaller so they are not our focus here. Nonetheless, our model can easily be extended to incorporate them.

[5]We only consider equality joins because the other types of joins cannot benefit from collocation.

[6]In cases of multi-way joins, the weight function evaluates the size of the intermediate tables.

[7]Appendix A.1 contains more details on how we chose this function for our experiments.

[8]Due to different filter conditions and updates of the database tables, the weight of the same join may differ among several queries.



(a) Example *Join Multi-Graph*. (b) $W_{R_{red}} = 6$.

(c) $W_{R_{green}} = 3$. (d) A real *Join Multi-Graph*.

Figure 2: Placement makes a difference: (a) example of a *Join Multi-Graph* and two DK recomendations, namely (b) 'red' setting saves 6 units of work (3 heavy, bold edges, have their tables collocated, e.g., $B.b_1 - D.d_1$); (c) 'green' setting saves only 3 units of work (d) real JMGs can be quite complex.

skew of the data distribution, meaning that a few compute nodes would be burdened by a large portion of the data, whereas other nodes would be practically empty. Handling of skew is outside the scope of our paper: if the data have skew, this will be reflected on the cost of the join, that is the weight $w(e)$, which is an input to our algorithm. In Appendix A.2 we provide more insight on our handling of skew for our experiments. In what follows, we assume that the input graph is already aware of skew-prone edges.

Figure 2(d) illustrates a real example of a *Join Multi-Graph*, in which for readability we have excluded all vertices with degree equal to one. Again the weight of an edge is proportional to its thickness. It is evident from the figure that the reality is often more complex than a star/snowflake schema, as many fact tables join each other on a plethora of attributes. Section 6 provides more insights about the typical characteristics of real *Join Multi-Graphs*.

In the next section we utilize our *Join Multi-Graph* data structure to formulate the *"Distribution-Key Recommendation"* problem.

3.2 Optimization problem: Distribution-Key Recommendation

Suppose we are given a query workload Q and the corresponding *Join Multi-Graph* $G_Q = (V, E, w)$. Let r_u be an element of u's attribute set, i.e., $r_u \in \mathcal{A}_u$. We refer to the pair $u.r_u$ as the *vertex recommendation* of u and we define the *total recommendation* R as the collection of all vertex recommendations , i.e., $R = \{u.r_u | u \in V\}$. For instance, the recommendation R that results in the distribution of Figure 1(c), is $R = \{(Customer.c_State), (Branch.b_State)\}$. Note that each vertex can have at most one vertex recommendation in R. We define the weight W_R of a total recommendation R as the sum of all weights of the edges whose endpoints belong to R, i.e.,

$$W_R = \sum_{\substack{e=(u.x,v.y)\\ r_u=x, r_v=y}} w(e) \qquad (1)$$

Intuitively the weight of a recommendation corresponds to the weight of the collocated joins that R would produce, if all tables $u \in R$ were distributed with $DK(u) = r_u$. Since we aim to collocate joins with maximum impact, our objective is to find the recommendation of maximum weight. Formally, the problem we are solving is as follows:

Problem 1 (*"Distribution-Key Recommendation"*). *Given a Join Multi-Graph G_Q, find the recommendation R^* with the maximum weight , i.e.,*

$$R^* = \operatorname*{argmax}_R W_R$$

Continuing the running example, Figures 2(b),(c) illustrate two different recommendations, namely R_{red} and R_{green}. A normal (resp. dashed) edge denotes a collocated (resp. non-collocated) join. $R_{red} = \{(B.b_1), (C.c), (D.d_1), (E.c), (F.c)\}$ and $R_{green} = \{(A.a), (B.b), (C.c), (D.d), (E.e), (F.a)\}$, with corresponding weights $W_{R_{red}} = w(C.c, F.c) + w(E.c, C.c) + w(B.b_1, D.d_1) = 6$ and $W_{R_{green}} = w(E.e, B.b) + w(B.b, D.d) + w(F.a, A.a) = 3$. Obviously R_{red} should be preferred as it yields a larger weight, i.e., more collocated joins. Our problem definition aims at discovering the recommendation with the largest weight (W_R) out of all possible combinations of vertex recommendations.

The *Join Multi-Graph* can be used to directly compare the collocation benefit between different recommendations. For instance, we can quickly evaluate a manual choice (e.g., choice made by a customer), and compare it to the result of our methods. If we deem that there is sufficient difference among the two, we can expose the redistribution scheme to the customers. Otherwise, if the difference is small, we should avoid the redistribution, which (depending on the size of the table) can be an expensive operation. When external factors make the potential future benefit of redistribution unclear (for instance, due to a possible change of architecture), the decision of whether or not to redistribute can be cast as an instance of the Ski Rental Problem [15, 27] (or related problems such as the Parking Permit Problem [18]), and therefore analyzed in the lens of competitive analysis, which is out of scope for this paper. Alternatively, one may resort to simple heuristics, e.g., by reorganizing only if the net benefit is above some threshold value relative to the table size.

Often times, the join attributes of a join either share the same label, or (more generally) they can be relabelled to a common label (without duplicating column names within any of the affected tables). For instance, in Figure 1, if the only joins are (`b_State`, `c_State`) and (`b_ID`, `c_ID`), we can relabel those columns to just '`State`' and '`ID`', respectively, without creating collisions in either table. The extreme case in which all joins in the query workload have this property commonly arises in discussion of related join-optimization problems (e.g. [10, 25, 26]) and is combinatorially interesting in its own right. We call this consistently-labelled subproblem $\mathrm{DKR}_{\mathrm{CL}}$.

PROBLEM 2 (DKR_{CL}). *Given a Join Multi-Graph G_Q in which all pairs of join attributes have the same label, find the recommendation R^* with the maximum weight, i.e., $R^* = \text{argmax}_R W_R$*

4. DKR - THEORY

In this section, we show that both variants of DKR are **NP**-complete, and are (to varying degrees) hard to approximate.

THEOREM 1 (HARDNESS OF DKR_{CL}). *DKR_{CL} generalizes the* MAXIMUM HAPPY EDGE COLORING *(*MHE*) problem, and is therefore both **NP**-complete and inapproximable to within a factor of* 951/952.

Proof. Refer to Section 2.2.2 for the definition of MHE. Given an MHE instance $\langle G = (V, E), \mathcal{C}, \varphi \rangle$, we can readily convert it into the framework of DKR_{CL} as follows. For each vertex $v \in V$ we create one table t_v. If v is in the domain of φ, then the attribute set of t_v is the singleton set $\{\varphi(v)\}$; otherwise, it is the entire color set $\mathcal{C}$. For each edge (u, v) of weight w, we add a join between t_u and t_v of that same weight for each color c in the attribute sets of both t_u and t_v (thus creating either 0, 1, or $|\mathcal{C}|$ parallel, equal-weight joins between the two tables). This completes the reduction.

An assignment of distribution keys to table t_v corresponds exactly to a choice of a color $\varphi'(v)$ in the original problem. Thus, solutions to the two problems can be put into a natural one-to-one correspondence, with the mapping preserving all objective values. Therefore, all hardness results for MAXIMUM HAPPY EDGE COLORING directly carry over to DKR_{CL}. ■

Intuitively, the above proof shows that MHE is a very particular special case of DKR. In addition to the assumptions defining DKR_{CL}, the above correspondence further limits the diversity of attribute sets (either to singletons or to all of $\mathcal{C}$), and requires that any two tables with a join between them must have an equal-weight join for every possible attribute (in both attribute sets). As we now show, the general problem without these restrictions is significantly harder.

THEOREM 2 (MAX-REP HARDNESS OF DKR). *There exists a polynomial time approximation-preserving reduction from* MAX-REP *to* DKR*. In particular,* DKR *is both **NP**-complete and inapproximable to within a factor of $2^{\log^{1-\epsilon} n}$ unless $\textbf{NP} \subseteq \textbf{DTIME}(n^{poly \log n})$, where n is the total number of attributes.*

Proof. Refer to Section 2.2.3 for the definition of MAX-REP. Given a MAX-REP instance with graph $G = (V, E)$, whose vertex set V is split into two collections of pairwise-disjoint subsets $A = \{A_1, A_2, \cdots\}$ and $B = \{B_1, B_2, \cdots\}$, we construct a DKR instance as follows. Let $\mathcal{C} = A \cup B$; our join multi-graph contains exactly one vertex u_{C_i} for each set $C_i \in \mathcal{C}$. Further, for each node u_{C_i}, we have one attribute $u_{C_i}.x$ for each $x \in C_i$. For each edge $(a, b) \in E$, we identify the sets $C_i \ni a$ and $C_j \ni b$ containing the endpoints and add a weight-1 edge between $u_{C_i}.a$ and $u_{C_j}.b$.

There is a one-to-one correspondence between feasible solutions to the initial MAX-REP instance and those of the constructed DKR instance. For any solution to the DKR instance, the selected hashing columns exactly correspond to picking vertices from the MAX-REP instance. Because each table is hashed on one column, the corresponding MAX-REP solution is indeed feasible. Similarly, the vertices chosen in any MAX-REP solution must correspond to a feasible assignment of distribution keys to each table. Because the objective function value is preserved by this correspondence, the theorem follows. ■

Next, we show that the approximability of DKR and MAX-REP differs by at most a logarithmic factor.

THEOREM 3. *An $f(n)$-approximation algorithm for* MAX-REP *can be transformed into an $O(f(n)(1+\log(\min(n, w_r))))$-approximation algorithm for* DKR*, where $w_r = w_{\max}/w_{\min}$ is the ratio of the largest to smallest nonzero join weights and n is the total number of attributes.*

Proof. We reverse the above reduction. Suppose we are given an instance of DKR with optimal objective value $\textbf{OPT}_{\textbf{DKR}}$. Without loss of generality, we can assume that all joins in the instance have weight at least $w_{\max}/n^3$; otherwise, we can discard all joins of smaller weight with only a sub-constant multiplicative loss in the objective score. Since $\log(w_{\max}/w_{\min}) = O(\log n)$ in this refined instance, $\log(\min(n, w_r)) = O(\log w_r)$ and therefore it suffices to find an algorithm with approx. guarantee $O(f(n)(1 + \log w_r))$.

We next divide all join weights by $w_{\min}$, and round down each join weight to the nearest (smaller) power of 2. The first operation scales all objective values by a factor of $1/w_{\min}$ (but otherwise keeps the structure of solutions the same) and the second operation changes the optimal objective value by a factor of at most 2. After this rounding, our instance has $k \leq 1 + \log_2 w_r$ different weights on edges. For each such weight w, consider the MAX-REP instance induced only on those joins of weight exactly w. Note that the sum of the optimal objective values for each of these k MAX-REP instances is at least equal to the objective value of the entire rounded instance, and is thus within a factor $1/(2w_{\min})$ of $\textbf{OPT}_{\textbf{DKR}}$. Thus, one of these k MAX-REP instances must have a solution of value at least $1/(2kw_{\min})$, and therefore our MAX-REP approximation algorithm can find a solution with objective value at least $\textbf{OPT}_{\textbf{DKR}}/(2kw_{\min}f(n))$. We output exactly the joins corresponding to this solution. In the original DKR instance, the weights of the joins in this output were each a factor of at least $w_{\min}$ greater than what we scaled them to, so the DKR objective value of our output is at least $\textbf{OPT}_{\textbf{DKR}}/(2kf(n))$. The conclusion follows from plugging in $k = O(1 + \log w_r)$. ■

As these theorems effectively rule out the possibility of exact polynomial-time algorithms for these problems, we study the efficacy of various efficient heuristic approaches, as well as the empirical running time of an exact super polynomial-time approach based on integer linear programming.

5. PROPOSED METHOD - ALGORITHMS

Based on the complexity results of the previous section, we propose two distinct types of methods to solve Problem 1: The first, is an exact method based on integer linear programming (Section 5.1). The second, is a collection of heuristic variants, which exploit some similarities of DKR related to graph matching (Section 5.2). Eventually, Section 5.3 presents our proposed meta-algorithm, namely BEST OF ALL WORLDS (BAW), a combination of the exact and heuristic approaches.

5.1 ILP: an exact algorithm

One approach to solving this problem involves integer programming. We construct the program as follows. We have one variable y_a for each attribute, and one variable x_{ab} for each pair (a, b) of attributes on which there exists a join. Intuitively, we say that $y_a = 1$ if a is chosen as the distribution key for its corresponding table, and $x_{ab} = 1$ if we select both a and b as distribution keys, and thus successfully manage to collocate their tables with respect to the join on a and b. To this end, our constraints ensure that each table selects exactly one attribute, as well as that the value of x_{ab} is bounded from above by both y_a and y_b (in an optimal solution, this means x_{ab} is 1 exactly when both y_a and y_b are 1). Finally, we wish to maximize the sum $\sum_{(a,b)\in E} w_{ab}x_{ab}$ of the captured join costs. The full integer program is provided below.

$$
\begin{aligned}
\text{maximize} \quad & \sum_{(a,b)\in E} w_{ab}x_{ab} & \\
\text{subject to} \quad & \sum_{a\in v} y_a = 1 & \forall v \in V, \\
& x_{ab} \le y_a & \forall a, b \in E, \\
& x_{ab} \le y_b & \forall a, b \in E, \\
& x_{ab}, y_a \in \{0, 1\} & \forall a, b \in E
\end{aligned} \tag{2}
$$

Unfortunately, there is no fast algorithm for solving integer linear programs unless $P = NP$. While the natural linear programming relaxation of this problem can be solved efficiently, it is not guaranteed to give integer values (or even half-integer values) to its variables, as implied by the inapproximability results of Section 4. Further, fractional solutions have no good physical interpretation in the context of DKR: only collocating half of the rows of a join does not result in half of the network cost savings of a full collocation of the same join. Thus, while we can run an ILP solver in the hope that it quickly converges to an integer solution, we also present a number of fast heuristic approaches for instances where the solver cannot quickly identify the optimum.

5.2 Heuristic algorithms

Here we describe our heuristic methods that are motivated by graph matching. We first discuss the relation of Problem 1 to matching and then we provide the algorithmic framework.

5.2.1 Motivation

Recall from Section 2.2.1 that the *Maximum Weight Matching* problem assumes as input a weighted graph $G = (V, E, w)$. The output is a set of edges $E_H \subseteq E$ such that each vertex $u \in V$ is incident to at most one edge in E_H.

Assume the *Join Multi-Graph* $G_Q = (V, E, w)$ of a query workload Q. We next investigate the relationship of Problem 1 to *Maximum Weight Matching*, starting from the *special case*, where the degree of any vertex $u \in V$ equals to the cardinality of u's attribute set[9], i.e., $\forall u \in V$, $deg_{G_Q}(u) = |\mathcal{A}_u|$. In other words, this assumption states that each join involving a table u is on different join attributes than all other joins that involve u. For instance, vertex F in Figure 2(a) has $deg_{G_Q}(F) = |\mathcal{A}_F| = |\{a, b, c\}| = 3$.

[9]We restrict our attention to the attributes that participate in joins only.

LEMMA 1. *Given a query workload Q and the corresponding Join Graph $G_Q = (V, E, w)$, assume a special case , where $deg_{G_Q}(u) = |\mathcal{A}_u|, \forall u \in V$. Then the solution to this* ***special case*** *of "Distribution-Key Recommendation" problem on Q is given by solving a Maximum Weight Matching on G_Q.*

Proof. A recommendation R is equivalent to the set of collocated edges. Since the degree of each vertex $u \in V$ equals to the cardinality of u's attribute set, all edges $E_u = e_1, e_2, \cdots e_{deg(u)} \subseteq E$ incident to u have distinct attributes of u, i.e, $e_i = (u.i, x.y)$. Then, any recommendation R contains at most one edge from E_u for a vertex u. Thus, R is a matching on G_Q. Finding the recommendation with the maximum weight is equivalent to finding a *Maximum Weight Matching* on G_Q. ■

In general, however, the cardinality of the attribute set of a vertex is not *equal* to its degree, but rather it is upper-bounded by it, i.e., $|\mathcal{A}_u| \le deg_{G_Q}(u), \forall u \in V$. With this observation and Lemma 1, we prove the following theorem:

THEOREM 4. *The Maximum Weight Matching problem is a special case of the "Distribution-Key Recommendation" problem.*

Proof. We show that any instance of *Maximum Weight Matching* can be reduced to an instance of *"Distribution-Key Recommendation"*. Let a weighted graph G be an input to *Maximum Weight Matching*. We can always construct a workload Q and the corresponding *Join Multi-Graph* G_Q according to the assumptions of Lemma 1. In particular, for any edge $e = (u, v, w)$ of G, we create an edge $e' = (u'.i, v'.j, w)$ ensuring that attributes i and j will never be used subsequently. The assumptions of Lemma 1 are satisfied, and an algorithm that solves *"Distribution-Key Recommendation"* would solve *Maximum Weight Matching*. The opposite of course does not hold, as shown in Section 4. ■

Motivated by the similarity to *Maximum Weight Matching*, we propose methods that first extract a matching and then they refine it. Currently, the fastest algorithms to extract a matching have complexity $O(\sqrt{V}E)$. This is too expensive for particularly large instances of the problem, where a strictly linear solution is required. For this reason, in the following we aim for approximate matching methods.

5.2.2 MATCH-N-GROW (MNG) algorithm

The main idea is to extract the recommendation in two phases. In Phase 1, MNG ignores the join attributes of the multigraph and extracts an approximate maximum weight matching in linear time, with quality guaranties that are at least 1/2 of the optimal [14]. In Phase 2, MNG greedily expands the recommendation of Phase 1.

Algorithm 1 contains the pseudocode of MNG. Phase 1 (lines 3-12) maintains two recommendations, namely R_1 and R_2 that are initially empty. At each iteration of the outer loop, it randomly picks a vertex $u \in V$ with degree at least one. Then, from all edges incident to u, it picks the *heaviest* one (i.e., $e = (u.x, v.y)$) and assigns the corresponding endpoints (i.e., $(u.x)$ and $(v.y)$) to R_i. The process is repeated for vertex v and recommendation R_{3-i} until no edge can be picked. Intuitively, Phase 1 extracts two *heavy alternating*

```
input  : a Join Multi-Graph G_Q = (V, E, w)
output: A distribution key recommendation R
1  R_1 ← ∅, R_2 ← ∅, E_r ← E, i ← 1
2  // PHASE 1: Maximal Matching
3  while E_r ≠ ∅ do
4  |   pick an active vertex u ∈ V with deg_u ≥ 1
5  |   while u has a neighbour do
6  |   |   e ← (u.x, v.y) such that w(e) ≥ w(e')∀e'
   |   |     incident to u // pick the heaviest edge
7  |   |   R_i ← R_i ∪ {(u, u.x), (v, v.y)}
8  |   |   i ← 3 − i // flip i : 1 ↔ 2
9  |   |   remove u and its incident edges from V
10 |   |   u ← v
11 |   end
12 end
13 //PHASE 2: Greedy Expansion
14 Let R be the recommendation with the max
     weight, i.e. R ← W(R_1) ≥ W(R_2)?R_1 : R_2
15 for u ∉ R do
16 |   e ← (u.x, v.y) such that w(e) ≥ w(e')∀e'
   |     incident to u with (v, v.y) ∈ R
17 |   R ← R ∪ {(u, u.x)}
18 end
19 return R
```

Algorithm 1: MNG: Heuristic approach based on maximum weight matching.

paths[10] and assigns their corresponding edges to recommendations R_1 and R_2 respectively. At the end of Phase 1, let recommendation R be the one with the largest weight.

Phase 2 (lines 15-18) extends the recommendation R by greedily adding more vertex recommendations; for each vertex u that does not belong to R, line 16 picks the heaviest edge (i.e., $e = (u.x, v.y)$) that is incident to u and has the other endpoint (i.e., $(v.y)$) in R. Then, it expands the recommendation by adding $(u.x)$. Since all edges are considered at most once, the complexity of Algorithm 1 is $O(E)$.

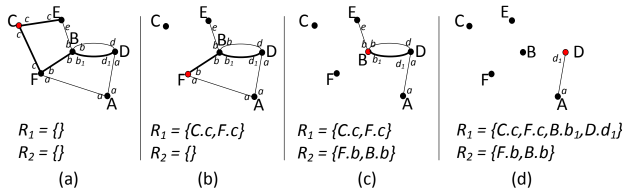


Figure 3: Example of Phase 1 of MNG. The method maintains two recommendations R_1 and R_2, which alternatively improves. The active vertex is red.

Figure 3 illustrates recommendations R_1 and R_2 on the *Join Multi-Graph* G_Q of Figure 2(a), during 4 iterations of the main loop of Phase 1 (i.e., lines 5-10). Initially, both recommendations are empty (Figure3(a)). Let C be the active vertex (marked with red). MNG picks the heaviest edge incident to C, i.e., $(C.c, F.c)$ and adds $(C.c), (F.c)$ in R_1, while removing all edges incident to C from the graph (Figure 3(b)). The active vertex now becomes F. Since $W(F.b, B.b) > W(F.a, A.a)$, R_2 becomes $(F.b, B.b)$ and the edges incident to F are removed (Figure 3(c)). Figure 3(d) illustrates the recommendations and the *Join Multi-Graph* at the end of Phase 1. R_1 is the heaviest of the two with $W(R_1) = 4 > W(R_2) = 2$, thus $R = R_1$. In Phase 2, vertex E picks the heaviest of its incident edges in G_Q (i.e., $(E.c, C.c)$) and the final recommendation becomes $R = \{(C.c), (F.c), (B.b_1), (D.d_1), (E.c)\}$, illustrated in Figure 2(b) with normal (non dashed) edges.

[10]Following the standard terminology of Matching theory, an *alternating path* is a path whose edges belong to matchings R_1 and R_2 alternatively.

Following the same two-phase approach, where Phase 1 performs a *maximal matching* and Phase 2 *greedy expansion*, we design a number of heuristics, described below. The following variations are alternatives to Phase 1 (Matching Phase) of Algorithm 1. They are all followed by the greedy expansion phase (Phase 2). All heuristics iteratively assign keys to the various tables, and never change the assignment of a table. Thus, at any point in the execution of an algorithm, we refer to the set of remaining *legal edges*, which are those edges $(u.x, v.y)$ whose set of two endpoint attributes $\{x, y\}$ is a superset of the DKs chosen for each of its two endpoint tables $\{u, v\}$ (and thus this join can still be captured by assigning the right keys for whichever of u and v is yet unassigned).

GREEDY_MATCHING *(*GM*).* Sort the tables in order of heaviest outgoing edge, with those with the heaviest edges coming first. For each table u in this ordering, sort the list of legal edges outgoing from u. For each edge $e = (u.x, v.y)$ in this list, assign key x to u and y to v (if it is still legal to do so when e is considered).

RANDOM_CHOICE *(*RC*).* Randomly split the tables into two sets, A_1 and A_2, of equal size (± 1 table). Take the better of the following two solutions: (a) For each $u \in A_1$, pick an attribute x at random. Once these attributes have all been fixed, for each $v \in A_2$, pick an attribute maximizing the sum of its adjacent remaining legal edges. (b) Repeat the previous process with the roles of A_1 and A_2 reversed.

RANDOM_NEIGHBOR *(*RN*).* For each attribute $u.x$ of any table, run the following procedure and eventually return the best result: 1) Assign x to u. 2) Let T be the set of tables adjacent to an edge with exactly one endpoint fixed. 3) For each table $v \in T$, let $e = (u.x, v.y)$ be the heaviest such edge, and assign key y to v. Repeat 2) and 3) until $T = \emptyset$.

Lastly, as a baseline to our experiments, we consider the naive greedy approach (Phase 2 of Algorithm 1).

NAIVE_GREEDY *(*NG*).* Start off with an empty assignment of keys to tables. Let E_r be the set of all legal joins. While E_r is non-empty, repeat the following three steps: (i) let $e = (u.x, v.y)$ be the maximum-weight edge in E_r, breaking ties arbitrarily; (ii) assign key x to u and v to y; (iii) remove e from E_r, as well as any other join in E_r made illegal by this assignment. Note that such a process will never assign two distinct keys to the same table. For any table that is not assigned a key by the termination of the above loop, assign it a key arbitrarily.

Notably, variants of heuristics GREEDY_MATCHING, RANDOM_CHOICE, and RANDOM_NEIGHBOR were previously studied in [4], where the authors proved various guarantees on their performance on MAX-REP instances. By alluding to the DKR–MAX-REP connection described in Section 4, one can expect similar quality guarantees to hold for our DKR instances.

5.3 BAW: a meta-algorithm

As we demonstrate in the experimental section, there is no clear winner among the heuristic approaches of Section 5.2. Thus, we propose BEST OF ALL WORLDS(BAW), a meta-algorithm that combines all previous techniques. It assigns a time budget to ILP and, if it times-out, it triggers all matching based variants of Section 5.2.2 and picks the one with the best score. Algorithm 2 contains the pseudo-code of BAW.

input : a JMG $G_Q = (V, E, w)$, a time budget t
output: A distribution key recommendation

1 $R_{\text{ILP}} \leftarrow$ ILP (G_Q) and kill after time t.
2 **if** *execution time of* ILP $\leq t$ **then**
3   **return** R_{ILP}
4 $R_{\text{MNG}} \leftarrow$ MNG (G_Q)
5 $R_{\text{GM}} \leftarrow$ GM (G_Q)
6 $R_{\text{RC}} \leftarrow$ RC (G_Q)
7 $R_{\text{RN}} \leftarrow$ RN (G_Q)
8 **return** $\max(R_{\text{ILP}}, R_{\text{MNG}}, R_{\text{GM}}, R_{\text{RC}}, R_{\text{RN}})$

Algorithm 2: BAW: *Best of all worlds* meta-algorithm.

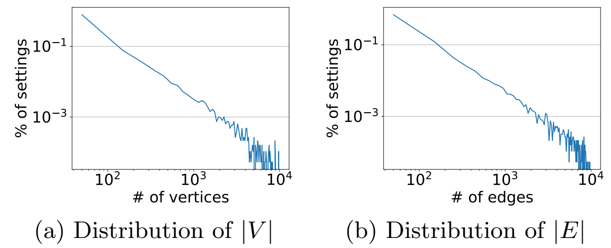
6. EXPERIMENTS

In our evaluation we use 3 real datasets: Real1 and Real2 are join graphs that are extracted at random from real life users of Redshift with various sizes and densities. Appendix A describes how edge weights were obtained. TPC-DS [19] is a well-known benchmark commonly applied to evaluate analytical workloads. In order to assess the behaviour of the methods in graphs with increasing density, we also use 4 synthetic *Join Multi-Graphs* created by randomly adding edges among 100 vertices (in a manner analogous to the Erdös-Réyni construction of random simple graphs [8]). The total number of edges ranges from 100 to 10 000, the maximum number of attributes per table ranges from 1 to 100, and all edge weights are chosen uniformly in the interval [0, 10 000]. Table 2 summarizes the characteristics of our datasets.

Table 2: Characteristics of datasets

dataset	vertices	edges	$\|E\|/\|V\|$
Real1	1 162	2 356	2.03
Real2	1 342	2 677	1.99
TPC-DS	24	126	5.25
Synthetic	100	100	1
		1 000	10
		5 000	50
		10 000	100

Figure 4 plots the distribution of size (in number of (a) vertices and (b) edges) of real Redshift Join Multi-Graphs



(a) Distribution of $|V|$ (b) Distribution of $|E|$

Figure 4: How big are join graphs in reality? Distribution of vertex-set and edge-set sizes.

in log-log scale. Note that as Appendix A.2 describes, edges that could cause data skew have been handled during a preprocessing phase. Both distributions follow the zipf curve with slope $\alpha = -2$: more than 99% of the graphs have less than 1000 vertices and edges, but there are still a few large graphs that span to several thousands of vertices and edges.

Section 6.1 compares our proposed methods in terms of their runtime and the quality of the generated recommendations. Section 6.2 demonstrates the effectiveness of our recommendations by comparing the workload of real settings, before and after the implementation of our recommended distribution keys.

6.1 Comparison of our methods

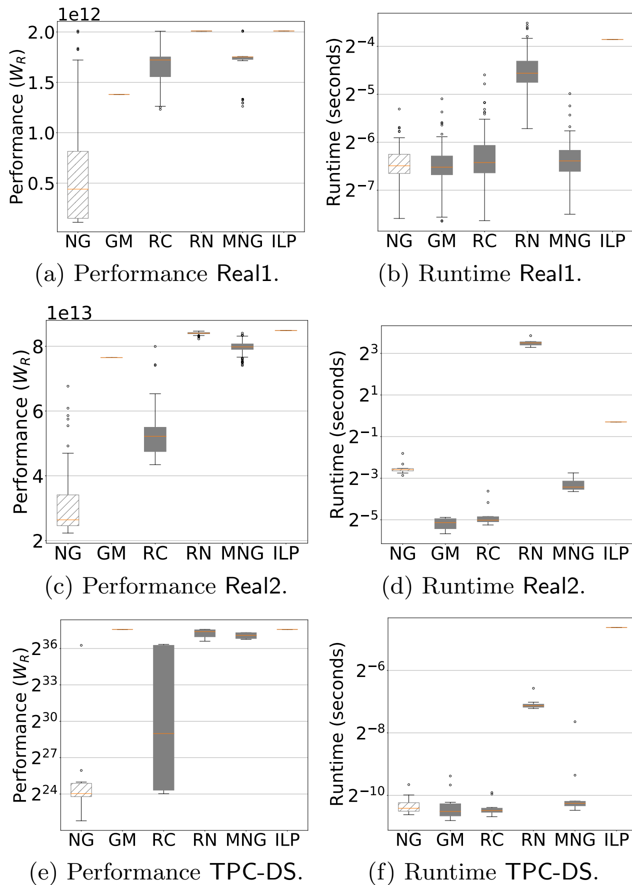
All methods were implemented in Python 3.6. For ILP we used PuLP 2.0 library with the CoinLP solver. We plot our experimental results using boxplots (y-axis). Specifically, each boxplot represents the distribution of values for a randomized experiment; the vertical line includes 95% of the values, the rectangle contains 50% of the values, and the horizontal line corresponds to the median value. The outliers are denoted with disks. For the randomized algorithms, we repeat each experiment 100 times. To visually differentiate the greedy baseline (NG) from the matching based heuristics, we use white hashed boxplot for NG and gray solid boxplots for the rest.

6.1.1 Real datasets

Figure 5 compares the performance and the runtime of our proposed methods on the real datasets. Each row of Figure 5 corresponds to a real dataset, namely Real1, Real2 and TPC-DS. The first column plots the performance of the various methods in terms of W_R, i.e., the total weight of the recommendation (the higher the better).

Overall, RN and MNG have very consistent performance that is closely approaching the exact algorithm (ILP). RN in particular, almost always meets the bar of the optimal solution. On the other hand, the greedy baseline NG varies a lot among the different executions, and usually underperforms: the mean value of NG is almost 3x worse than the optimal. This is expected as greedy decisions are usually suboptimal because they only examine a very small subset of the graph and they do not account for the implications to the other vertices. In very few cases (around 1%) NG manages to return a solution with weight similar to the optimal.

The second column of Figure 5 plots the runtime comparison of our methods (the lower the better). Note that the y-axis is in logarithmic scale. Notably, RN takes much longer than the other heuristics to complete (its mean value



(a) Performance Real1. (b) Runtime Real1.

(c) Performance Real2. (d) Runtime Real2.

(e) Performance TPC-DS. (f) Runtime TPC-DS.

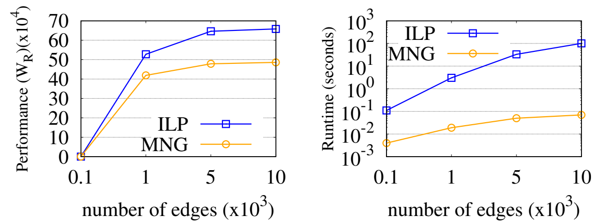
Figure 5: Performance and runtime of methods real datasets. No clear winner.

is 4x higher than the other techniques), and its runtime is comparable to only that of ILP, which yields the exact solution. This indicates that the consistently high performance of the method comes at a high cost. Concerning the other heuristics, they all perform similarly in terms of runtime, with GM having a slight edge over the others. Overall, MNG is consistently highly performant and runs very fast.

6.1.2 Synthetic datasets

To evaluate the scalability of the various methods, we utilize synthetic workloads, with a varying number of edges. In all cases, the number of nodes is fixed to 100. As a representative technique of the matching approaches, we focus our attention on MNG. NG has already shown to underperform in our real datasets, so we exclude it from further consideration.

Figure 6(a) compares the two methods on the quality of the returned solution. The x-axis corresponds to the size (in thousands of edges) of the synthetic graphs and the y-axis corresponds to W_R, i.e., the total weight of the recommendation. ILP finds a better solution compared to MNG, especially for denser graphs, for which it can be up to 20% better. Figure 6(b) compares the execution time of the two approaches. ILP is always slower than MNG, because the first gives the exact solution, whereas the second heuristically finds an approximate. ILP's runtime increases exponentially with the density of the graph (reaching 100 seconds



(a) Performance. (b) Runtime.

Figure 6: Results on synthetic datasets. (a) Performance as weight of recommendation (W_R) of ILP and MNG. (b) Runtime of ILP and MNG (seconds).

for the densest), whereas MNG is always much faster and stays under 0.1 seconds even for dense graphs (10k edges).

Our combined meta-algorithm BAW finds a balance between the runtime and quality of the proposed techniques by applying a time budget to the execution of ILP. Given that the size of the largest connected components of real join graphs is typically within an order of magnitude of a thousand edges, BAW finds the *optimal solution* in a matter of seconds in Amazon Lambda (the allowed time budget t was 5 minutes but ILP finishes in less than a minute in the vast majority of the cases).

6.2 Quality of recommendations

This section evaluates the actual *benefit* (in terms of query execution and network cost) of *"Dist-Key"* distribution style using the keys recommended by BAW, compared to the default *"Even"* distribution, i.e., Round Robin distribution of the table rows among the nodes of the cluster.

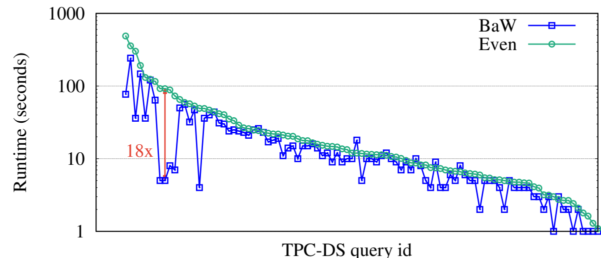
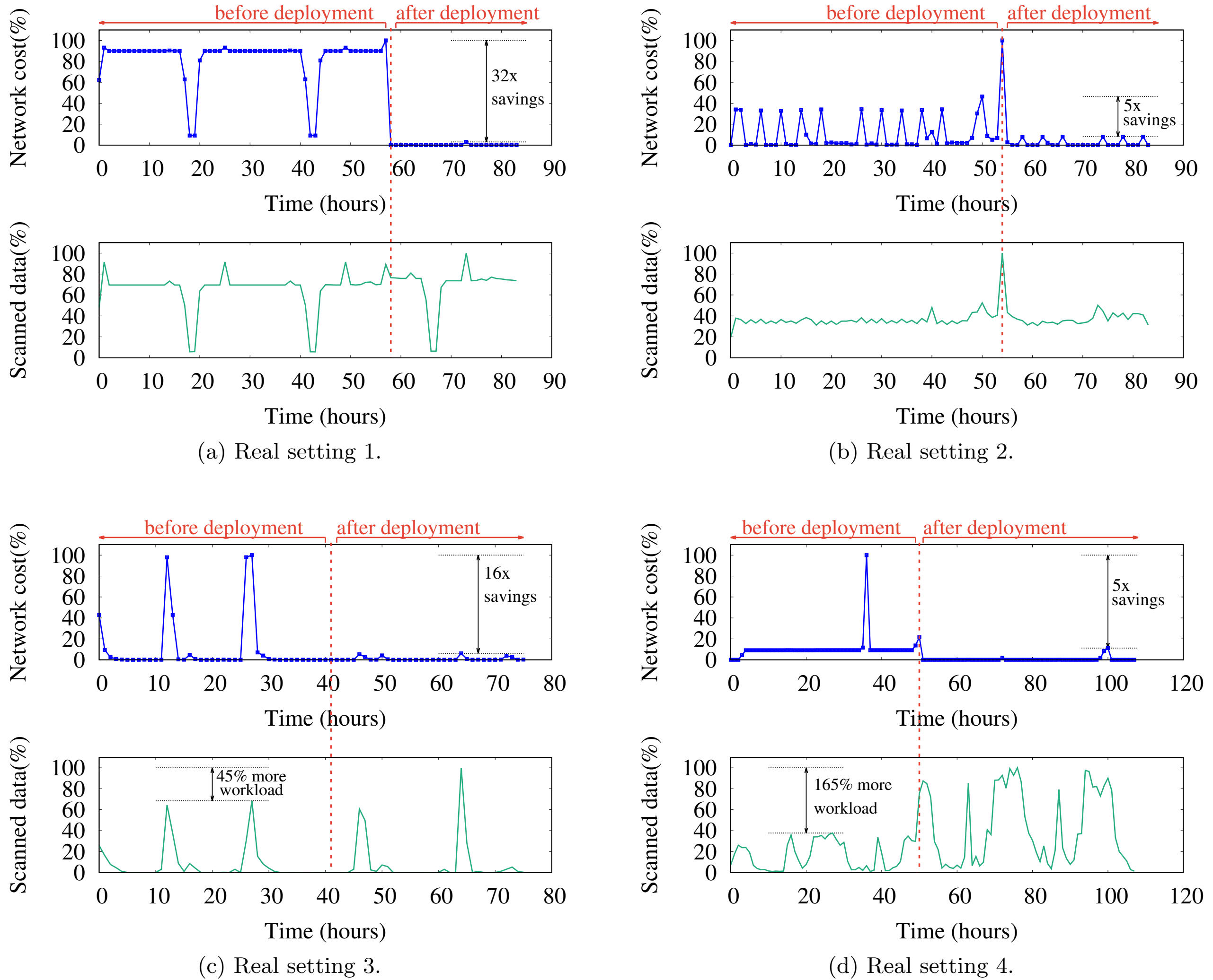


Figure 7: BAW wins almost always: Quality of recommendations in TPC-DS. Running time (log scale) vs query-ID - BAW often achieves dramatic savings (18x) especially at the heavier queries.

Initially we perform the experiment in 3TB TPC-DS workload, using a four-node *dc2.8xl* Redshift cluster [12]. We run the workload once and we construct the corresponding *Join Multi-Graph*. We use BAW to pick the best distribution keys for the observed workload. With these keys, we distribute the tables' data using Redshift's Dist-Key distribution style[11]. Finally, we run again the workload to compare the runtime of each of the queries among the two settings (in

[11]We used the default *"Even"* distribution if BAW produced no recommendation for a certain table.



(a) Real setting 1.
(b) Real setting 2.
(c) Real setting 3.
(d) Real setting 4.

Figure 8: BaW saves: Effect of recommendations on real data. [upper-blue line] Network cost (%) for queries touching the recommended tables. The red line indicates the time of implementation of our recommendations. [lower-green line] Total scanned data (%) versus time for the redistributed tables.

both cases we discard the first run to exclude compilation time).

Figure 7 (alternative representation of Figure 1(d)) illustrates the execution time (log scale) versus the query id for the two methods, where the queries are ordered by their runtime in *"Even"*. For the majority of the queries, BaW outperforms *"Even"* by up to 18x, as it is illustrated by the red arrow in the figure. This enormous performance benefit is explained by the collocation of expensive joins and shows that BaW is capable of finding the most appropriate distribution keys for complex datasets such as TPC-DS. On the other hand, several queries perform similarly in the two distribution styles because they are not joining any of the redistributed tables. Finally, we see some regression towards the tail of the distribution, i.e., for queries that run in under 10 seconds; this is attributed to the fact that Dist-Key may introduce slight skew in the data distribution, which can marginally regress some already fast queries. However, the small overhead is insignificant relative to the huge benefit of collocation for expensive queries[12].

We repeat this experiment on real-world workloads of users that implemented our recommendations (Figure 8). We picked 4 real scenarios and observed the workload for a few days. Based on our observation we incrementally generate the *Join Multi-Graph* and we run BaW to recommend the most appropriate distribution keys to some target tables. Data redistribution of all target tables happens at approximately the same time. For all queries that touch these tables, we keep track of the total network cost (i.e., data broadcasted or distributed at query time) and the total workload (i.e., data scanned at query time), before and after the data redistribution.

[12]Note that further performance improvement in TPC-DS is achieved by also picking the most appropriate sort-keys [13].

Each sub-figure of Figure 8 corresponds to one of our real settings and consists of two plots; the upper one - *blue line* illustrates the total network cost and the lower one -*green line* the total workload (data scanned) of relevant tables over the same period of time. The x-axis corresponds to the hours around the time of deployment, which is indicated by a red, dotted, vertical line.

Overall we observe that after deployment, the network cost drops by as much as **32x** (Figures 8(a), (b)), whereas the total workload (data scanned) for the same period of time remains stable. This clearly shows that BAW's choice of distribution keys was very effective: the query workload did not change, but the network cost dropped drastically. In addition, many of the joins that used to require data redistribution or broadcasting at query time now happen locally to each compute node, which explains why the cost drops to zero at times.

In Figure 8(b), around the time of deployment, we see a spike in both the network cost and the scanned data. This is expected because changing the distribution key of the tables itself may induce network cost that could be considerable, depending on the size of the tables. Nonetheless, this is a one time effort whereas the benefits of collocation accumulate over time.

In addition, Figures 8(c),(d) show a drastic drop of the network cost ($16x$ and $5x$ respectively), even when the total amount of scanned data *increases* by up to $2.7x$. The reason for this[13] is that, without the network overhead, queries are more efficient, and thus they use system resources for less time. Consequently, execution slots are released faster allowing more queries to run in the unit of time, resulting in increased scanned data of the relevant tables.

To summarize, the observations of our experiments are:

- **Very fast techniques**: The majority of the heuristic methods terminate within seconds for graphs of thousands of edges. ILP in Amazon Lambda finds the optimal solution in less than a minute for almost all cases seen in Redshift so far.
- **No clear winner among heuristics**: The heuristic methods perform differently in our various settings. This motivates BAW, a meta-algorithm that picks the best of the individual techniques.
- **Network cost reduction**: BAW applied in 4 real scenarios identified distribution keys that decreased the network cost up to **32x** for joins over constant or even heavier query workload.

7. CONCLUSIONS

In this paper we formulate the problem of *"Distribution-Key Recommendation"* as a novel combinatorial optimization problem defined on the *Join Multi-Graph*, and we study its complexity. We show that not only it is **NP**-complete, but it is also hard to approximate within any constant factor. For this reason, we present BAW, a meta-algorithm to solve *"Distribution-Key Recommendation"* problem efficiently. Its main idea is to combine an exact ILP based solution, with heuristic solutions based on maximum weight matching. The exact solution is only allowed to run for a given time budget; the heuristic algorithms follow a two phase approach: Phase 1 performs an approximate maximum matching on *Join Multi-Graph* and Phase 2, greedily extends the recommendation. BAW picks the best of the above methods.

The contributions of our approach are:

- **Problem formulation**: We introduce *Join Multi-Graph*, a concise graph representation of the join history of a cluster and we define the *"Distribution-Key Recommendation"* problem (DKR) as a novel graph problem on the *Join Multi-Graph*.
- **Theoretical Analysis**: We analyse the complexity of DKR and of some special case variants and we prove that they are both **NP**-complete and hard to approximate.
- **Fast Algorithm**: We propose BAW, an efficient meta-algorithm to solve DKR.
- **Validation on real data**: We experimentally demonstrate that BAW improves the performance of real workloads by virtually eliminating the network cost.

[13]Based on the assumption that the workload remains constant.

APPENDIX

A. IMPLEMENTATION DETAILS

This section discusses some implementation details used in our experiments, concerning the selection of weights and the handling of skew for edges of the *Join Multi-Graph*. They are included here for completeness as these decisions are orthogonal to the problem we solve in this paper.

A.1 Choosing weights

Recall from Section 3.2 that the weight $w(e) : E \to \mathbb{N}^+$ of an edge e represents the cumulative number of bytes processed by that join (after any pre-join filters and column pruning has been applied). For instance, assume b_1 and b_2 the bytes of join tables t_1 JOIN t_2, and let $b_1 < b_2$. The weight of edge $(t_1.b_1, t_2.b_2)$ was given as an input based on the formula:

$$W = \min\left(b_1 + b_2, b_1 * NODES\right) \tag{3}$$

where $NODES$ is the number of nodes in the cluster. The two arguments of min() in Equation 3 correspond to the cost of the two options of Redshift's optimizer to perform the distributed hash join (for non collocated tables): 1) to redistribute both tables on the join attributes or 2) to broadcast the smaller table to all compute nodes. Intuitively, the above function approximates the communication cost that one would have to pay during query execution, if the tables were not collocated. Note that b_1 and b_2 can be obtained asynchronously from the system logs of a join, thus there is no overhead in query execution.

A.2 Handling of data skew

We assume that penalty for data skew is already included in the join cost (see Section 3.2 Equation 1), which is an input to our problem, and thus orthogonal to the method proposed in this work. For our experiments, columns that may introduce data skew are penalized using statistics: if the number of distinct values of a column over the approximate total number of rows in the table is larger than a threshold $\theta1$, and the number of distinct values over the number of compute nodes is larger than a threshold $\theta2$, then the column contains enough distinct values to be considered safe in terms of skew. Otherwise it is removed from the graph. Thresholds $\theta1$, $\theta2$ were chosen experimentally.

8. REFERENCES

[1] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD*, pages 359–370, 2004.

[2] N. Borić, H. Gildhoff, M. Karavelas, I. Pandis, and I. Tsalouchidou. Unified spatial analytics from heterogeneous sources with Amazon Redshift. In *SIGMOD*, pages 2781–2784, 2020.

[3] M. Cai, M. Grund, A. Gupta, F. Nagel, I. Pandis, Y. Papakonstantinou, and M. Petropoulos. Integrated querying of SQL database data and S3 data in Amazon Redshift. *IEEE Data Eng. Bull.*, 41(2):82–90, 2018.

[4] M. Charikar, M. Hajiaghayi, and H. Karloff. Improved approximation algorithms for label cover problems. In *European Symposium on Algorithms*, pages 23–34. Springer, 2009.

[5] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: A workload-driven approach to database replication and partitioning. *PVLDB*, 3(1-2):48–57, 2010.

[6] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The complexity of multiterminal cuts. *SIAM Journal on Computing*, 23(4):864–894, 1994.

[7] G. Eadon, E. I. Chong, S. Shankar, A. Raghavan, J. Srinivasan, and S. Das. Supporting table partitioning by reference in oracle. In *SIGMOD*, pages 1111–1122, 2008.

[8] P. Erdös, A. Rényi, et al. On random graphs. *Publicationes mathematicae*, 6(26):290–297, 1959.

[9] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[10] P. Furtado. Workload-based placement and join processing in node-partitioned data warehouses. In *International Conference on Data Warehousing and Knowledge Discovery*, pages 38–47. Springer, 2004.

[11] C. Garcia-Alvarado, V. Raghavan, S. Narayanan, and F. M. Waas. Automatic data placement in MPP databases. In *IEEE 28th International Conference on Data Engineering Workshops*, pages 322–327, 2012.

[12] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan. Amazon Redshift and the case for simpler data warehouses. In *SIGMOD*, pages 1917–1923, 2015.

[13] J. Harris et al. Optimized DDL for TPC-DS in Amazon Redshift, 2020. `https://github.com/awslabs/amazon-redshift-utils/tree/master/src/CloudDataWarehouseBenchmark/Cloud-DWB-Derived-from-TPCDS/3TB` [Online; accessed 15-July-2020].

[14] S. Hougardy. Linear time approximation algorithms for degree constrained subgraph problems. In *Research Trends in Combinatorial Optimization*, pages 185–200. Springer, 2009.

[15] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1-4):79–119, 1988.

[16] G. Kortsarz. On the hardness of approximating spanners. *Algorithmica*, 30(3):432–450, 2001.

[17] M. Langberg, Y. Rabani, and C. Swamy. Approximation algorithms for graph homomorphism problems. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 176–187. Springer, 2006.

[18] A. Meyerson. The parking permit problem. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 274–282, 2005.

[19] R. O. Nambiar and M. Poess. The making of TPC-DS. In *VLDB*, pages 1049–1058, 2006.

[20] R. Nehme and N. Bruno. Automated partitioning design in parallel database systems. In *SIGMOD*, pages 1137–1148, 2011.

[21] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD*, pages 61–72, 2012.

[22] J. Rao, C. Zhang, N. Megiddo, and G. Lohman. Automating physical database design in a parallel database. In *SIGMOD*, pages 558–569, 2002.

[23] D. Sacca and G. Wiederhold. Database partitioning in a cluster of processors. *ACM Transactions on Database Systems (TODS)*, 10(1):29–56, 1985.

[24] T. Stöhr, H. Märtens, and E. Rahm. Multi-dimensional database allocation for parallel data warehouses. In *VLDB*, pages 273–284, 2000.

[25] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. MapReduce and parallel DBMSs: friends or foes? *Communications of the ACM*, 53(1):64–71, 2010.

[26] R. Varadarajan, V. Bharathan, A. Cary, J. Dave, and S. Bodagala. DBDesigner: A customizable physical design tool for Vertica analytic database. In *ICDE*, pages 1084–1095, 2014.

[27] Wikipedia contributors. Ski rental problem — Wikipedia, the free encyclopedia, 2020. `https://en.wikipedia.org/w/index.php?title=Ski_rental_problem&oldid=954059314` [Online; accessed 13-May-2020].

[28] E. Zamanian, C. Binnig, and A. Salama. Locality-aware partitioning in parallel database systems. In *SIGMOD*, pages 17–30, 2015.

[29] P. Zhang, Y. Xu, T. Jiang, A. Li, G. Lin, and E. Miyano. Improved approximation algorithms for the maximum happy vertices and edges problems. *Algorithmica*, 80:1412–1438, 2018.

[30] D. C. Zilio, A. Jhingran, and S. Padmanabhan. Partitioning key selection for a shared-nothing parallel database system. *IBM Research Report*, RC 19820, 1994.