Fast Join Project Query Evaluation using Matrix Multiplication

Shaleen Deep
University of Wisconsin-Madison
shaleen@cs.wisc.edu

Xiao Hu
Duke University
xh102@cs.duke.edu

Paraschos Koutris
University of Wisconsin-Madison
paris@cs.wisc.edu

ABSTRACT

In the last few years, much effort has been devoted to developing join algorithms to achieve worst-case optimality for join queries over relational databases. Towards this end, the database community has had considerable success in developing efficient algorithms that achieve worst-case optimal runtime for full join queries, i.e., joins without projections. However, not much is known about join evaluation with *projections* beyond some simple techniques of pushing down the projection operator in the query execution plan. Such queries have a large number of applications in entity matching, graph analytics and searching over compressed graphs. In this paper, we study how a class of join queries with projections can be evaluated faster using worst-case optimal algorithms together with matrix multiplication. Crucially, our algorithms are parameterized by the output size of the final result, allowing for choosing the best execution strategy. We implement our algorithms as a subroutine and compare the performance with state-of-the-art techniques to show they can be improved upon by as much as 50x. More importantly, our experiments indicate that matrix multiplication is a useful operation that can help speed up join processing owing to highly optimized open source libraries that are also highly parallelizable.

ACM Reference Format:

1 INTRODUCTION

In this paper, we study the problem of evaluating join queries where the join result does not contain all the variables in the body of the query. In other words, some of the variables have been *projected out* of the join result. The simplest way to evaluate such a query is first to compute the full join, make a linear pass over the result, project each tuple, and finally remove the duplicates. While this approach is conceptually simple, it relies on efficient *worst-case optimal join algorithms* [12, 32, 33, 35]. The main result of this line of work is a class of algorithms that run in time $O(|\mathbf{D}|^{\rho^*} + |\text{OUT}|)$, where $\mathbf{D}$ is the database instance and ρ^* is the optimal fractional edge cover of the query [12]. In the worst case, there exists a database $\mathbf{D}$ with $|\text{OUT}| = |\mathbf{D}|^{\rho^*}$. In practice, most query optimizers create a query plan by pushing down projections in the join tree.

Example 1. *Consider relation $R(x, y)$ of size N that represents a social network graph where an edge between two users x and y denotes that x and y are friends. We wish to enumerate all users pairs who have at least one friend in common [31]. This task can be captured by the query $\ddot{Q}(x, z) = R(x, y), R(z, y)$, which corresponds to the following* SQL *query:*

```
SELECT DISTINCT R1.x, R2.x
FROM R1 as R, R2 as R WHERE R1.y = R2.y
```

Suppose that the graph contains a small (constant) number of communities and the users are spread evenly across them. Each community has $O(\sqrt{N})$ users, and there exists an edge between most user pairs within the same community. In this case, the full join result is $\Theta(N^{3/2})$ but $|\ddot{Q}(\mathbf{D})| = \Theta(N)$.

As the above example demonstrates, using worst-case optimal join algorithms can lead to an intermediate output that can be much larger than the final result after projection, especially if there are many duplicate tuples. Thus, we ask whether it is possible to design faster algorithms that skip the construction of the full result when it is large, and as a result, speed up the evaluation. Ideally, we would like to have algorithms that run faster than worst-case optimal join algorithms, are sensitive to the output of the projected result, and have low memory requirements during execution.

In this paper, we show how to achieve the above goal for a fundamental class of join queries called *star joins*. Star joins are join queries where every relation joins on the same

variable. The motivation to build faster algorithms for star joins with projection is not limited to faster query execution in DBMS systems. Next, we present a list of three applications that benefit from these faster algorithms.

Set Similarity. Set similarity is a fundamental operation in many applications such as entity matching and recommender systems. Here, the goal is to return all pairs of sets such that contain at least c common elements. Recent work [21] gave the first output-sensitive algorithm that enumerates all similar sets in time $O(|\mathbf{D}|^{2-\frac{1}{c}} \cdot |\mathsf{OUT}|^{\frac{1}{2c}})$. As the value of c increases, the running time tends to $O(|\mathbf{D}|^2)$. The algorithm also requires $O(|\mathbf{D}|^{2-\frac{1}{c}} \cdot |\mathsf{OUT}|^{\frac{1}{2c}})$ space. We improve the running time and the space requirement of the algorithm for a large set of values that $|\mathsf{OUT}|$ can take, for all c.

Set Containment. The efficient computation of set containment joins over set-valued attributes has been extensively studied in the literature. A long line of research [27, 29, 30, 40] has developed trie-based join methods where the algorithm performs an efficient blocking step. We show that for certain datasets, our algorithm can identify set containment relationships much faster than state-of-the-art techniques.

Graph Analytics. In the context of graph analytics, the graph to be analyzed is often defined as a *declarative* query over a relational schema [7, 36, 38, 39]. For instance, consider the DBLP dataset, which stores author and paper pairs in a table $R(author, paper)$. To analyze the relationships between co-authors, we can extract the *co-author graph*, which we can express as the view $V(x, y) = R(x, p), R(y, p)$. Recent work [36] has proposed compression techniques where a preprocessing step generates a succinct representation of $V(x, y)$. However, these techniques require a very expensive pre-processing step, rely on heuristics, and do not provide any formal guarantees on the running time. Further, suppose that we want to support an API where a user can check whether authors a_1 and a_2 have co-authored a paper or not. This is an example of a *boolean query*. In this scenario, the view $R(x, p), R(y, p)$ is implicit and not materialized. Since such an API may handle thousands of requests per second, it is beneficial to batch B queries together and evaluate them at once. We show that our algorithms can lead to improved performance by minimizing user latency and resource usage.

Our contribution. In this paper, we show how to evaluate star join queries with projections using output-sensitive algorithms. We summarize our technical contribution below.

(1) Our main contribution (Section 3) is an output-sensitive algorithm that evaluates star join queries with projection. We use worst-case optimal joins and matrix multiplication as two fundamental building blocks to split the join into multiple subjoin queries which are evaluated separately. This technique was initially introduced by [11], but their runtime analysis is incorrect for certain regimes of the output size. We improve and generalize the results via a more careful application of (fast) matrix multiplication.

(2) We show (Section 4) how to exploit the join query algorithms for the problems of set similarity, set containment, join processing and boolean set intersection. Our algorithms also improve the best known preprocessing time bounds for creating offline data structures for the problems of set intersection [20] and compressing large graphs [36].

(3) We develop a series of optimization techniques that address the practical challenges of incorporating matrix multiplication algorithms to join processing.

(4) We implement our solution as an in-memory prototype and perform a comprehensive benchmarking to demonstrate the usefulness of our approach (Section 6). We show that our algorithms can be used to improve the running time for set similarity, set containment, join processing and boolean query answering over various datasets for both single-threaded and multi-threaded settings. Our experiments indicate that matrix multiplication can achieve an order of magnitude speedup on average and up to 50× speedup over state-of-the-art baselines.

2 PROBLEM SETTING

In this section, we present the basic notions and terminology and then define the problems we study in this paper.

2.1 Problem Definitions

In this paper, we will focus on the 2-path query, which consists of a binary join followed by a projection:

$$\ddot{Q}(x, z) = R(x, y), S(z, y)$$

and its generalization as a *star join*:

$$Q_k^{\star}(x_1, x_2, \dots, x_k) = R_1(x_1, y), R_2(x_2, y), \dots, R_k(x_k, y).$$

We will often use the notation $\mathbf{dom}(x)$ to denote the constants that the values variable x can take. We use $Q(\mathbf{D})$ to denote the result of the query Q over input database $\mathbf{D}$, or also OUT when it is clear from the context. Apart from the above queries, the following closely related problems will also be of interest.

Set Similarity (SSJ). In this problem, we are given two families of sets represented by the binary relations $R(x, y)$ and $S(z, y)$. Here, $R(x, y)$ means set x contains element y, and $S(z, y)$ means set z contains element y. Given an integer $c \geq 1$, the set similarity join is defined as

$$\{(a, b) \mid |\pi_y(\sigma_{x=a}(R)) \cap \pi_y(\sigma_{z=b}(S))| \geq c\}$$

In other words, we want to output the pairs of sets where the intersection size is at least c. When $c = 1$, SSJ becomes equivalent to the 2-path query $\ddot{Q}$. The generalization of set similarity to more than two relations can be defined in a similar fashion. Previous work [21] only considered the unordered version of SSJ. The *ordered* version simply enumerates OUT in decreasing order of similarity. This allows users to see the most similar pairs first instead of enumerating output tuples in arbitrary order.

Set Containment (SCJ). Similar to SSJ, given two families of sets represented by the relations R, S, we want to output

$$\{(a, b) \mid \pi_y(\sigma_{x=a}(R)) \subseteq \pi_y(\sigma_{z=b}(S))\}$$

In other words, we want to output the pairs of sets where one set is contained in the other.

Boolean Set Intersection (BSI). In this problem, we are given again two families of sets represented by the relations R, S. Then, for every input pair two sets a, b, we want to answer the following boolean CQ which asks whether the two sets have a non-empty intersection: $Q_{ab}() = R(a, y), S(b, y)$. If we also want to output the actual intersection, we can use the slightly modified CQ $\bar{Q}_{ab}(y) = R(a, y), S(b, y)$, which does not project the join variable. The boolean set intersection problem has been a subject of great interest in the theory community [10, 17, 18, 20, 34] given its tight connections with distance oracles and reachability problems in graphs.

In order to study the complexity of our algorithms, we will use the uniform-cost RAM model [25], where data values, as well as pointers to databases, are of constant size. Throughout the paper, all complexity results are with respect to data complexity where the query is assumed fixed.

2.2 Matrix Multiplication

Let A be a $U \times V$ matrix and C be a $V \times W$ matrix over a field $\mathcal{F}$. We use $A_{i,j}$ to denote the entry of A located in row i and column j. The matrix product AC is a $U \times W$ matrix with entries $(AC)_{i,j} = \sum_{k=1}^{V} A_{i,k} C_{k,j}$.

Join-Project as Matrix Multiplication. It will be convenient to view the 2-path query as a matrix computation operation. Let $\mathcal{A}, \mathcal{B}$ be the adjacency matrices for relations R, S respectively: this means that $\mathcal{A}_{i,j} = 1$ if and only if tuple $(i, j) \in R$ (similarly for S). Observe that although each relation has size at most $|\mathbf{D}|$, the input adjacency matrix can be as large as $|\mathbf{D}|^2$. The join output result $\ddot{Q}(D)$ can now be expressed as the matrix product $\mathcal{A} \cdot \mathcal{B}$ where matrix multiplication is performed over the boolean field.

Complexity. Multiplying two square matrices of size n trivially takes time $O(n^3)$, but a long line of research on fast matrix multiplication has dropped the complexity to $O(n^\omega)$, where $2 \leq \omega < 3$. The current best known value is $\omega = 2.373$ [22], but it is believed that the actual value is $2 + o(1)$.

We will frequently use the following folklore lemma.

Lemma 1. *Let ω be any constant such that we can multiply two $n \times n$ matrices in time $O(n^\omega)$. Then, two matrices of size $U \times V$ and $V \times W$ can be multiplied in time $M(U, V, W) = O(UVW\beta^{\omega-3})$, where $\beta = \min\{U, V, W\}$.*

For the theoretically best possible $\omega = 2+o(1)$, rectangular fast matrix multiplication can be done in time $O(UVW/\beta)$.

2.3 Known Results

Ideally, we would like to compute $Q_k^\star$ in time linear to the size of the input and the output. However, [13] showed that $\ddot{Q}$ cannot be evaluated in time $O(|\text{OUT}|)$ assuming that the exponent ω in matrix multiplication is greater than two.

A straightforward way to compute any join query with projections is to compute the join using any worst-case optimal algorithm and then deduplicate to find the projection. This gives the following baseline result.

Proposition 1 ([32, 33]). *Any CQ Q with optimal fractional edge cover ρ^* can be computed in time $O(|\mathbf{D}|^{\rho^*})$.*

Proposition 1 implies that we can compute the star query $Q_k^\star$ in time $O(|\mathbf{D}|^k)$, where k is the number of joins. However, the algorithm is oblivious to the actual output OUT and will have the same worst-case running time even when OUT is much smaller than $|\mathbf{D}|^k$ – as it often happens in practice. To circumvent this issue, [11] showed the following output sensitive bound that uses only combinatorial techniques:

Lemma 2 ([11]). *$Q_k^\star$ can be computed in time $O(|\mathbf{D}| \cdot |\text{OUT}|^{1-\frac{1}{k}})$.*

For $k = 2$, the authors make use of fast matrix multiplication to improve the running time to $\tilde{O}(N^{0.862} \cdot |\text{OUT}|^{0.408} + |\mathbf{D}|^{2/3} \cdot |\text{OUT}|^{2/3})$. In the next section, we will discuss the flaws in the proof of this result in detail.

3 COMPUTING JOIN-PROJECT

In this section, we describe our main technique and its theoretical analysis.

3.1 The 2-Path Query

Consider the query $\ddot{Q}(x, z) = R(x, y), S(z, y)$. Let N_R and N_S denote the cardinality of relations R and S respectively. Without loss of generality, assume that $N_S \leq N_R$. For now, assume that we know the output size $|\text{OUT}|$; we will show how to drop this assumption later.

We will also assume that we have removed any tuples that do not contribute to the query result, which we can do during a linear time preprocessing step.

Algorithm. Our algorithm follows the idea of partitioning the input tuples based on their degree as introduced in [11], but it differs on the choice of threshold parameters. It is

Algorithm 1: Computing $\pi_{xz} R(x,y) \bowtie S(z,y)$

---

1   $R^- \leftarrow \{R(a,b) \mid |\sigma_{x=a}R(x,y)| \leq \Delta_2$ or $|\sigma_{y=b}S(z,y)| \leq \Delta_1\}$, $R^+ \leftarrow R \setminus R^-$

2   $S^- \leftarrow \{S(c,b) \mid |\sigma_{z=c}S(z,y)| \leq \Delta_2$ or $|\sigma_{y=b}S(z,y)| \leq \Delta_1\}$, $S^+ \leftarrow S \setminus S^+$

3   $T \leftarrow (R^- \bowtie S) \cup (R \bowtie S^-)$ /* use wcoj */

4   $M_1(x,y) \leftarrow R^+$ adj matrix, $M_2(y,z) \leftarrow S^+$ adj matrix

5   $M \leftarrow M_1 \times M_2$ /* matrix multiplication */

6   $T \leftarrow T \cup \{(a,c) \mid M_{ac} > 0\}$

7   **return** T

---

parametrized by two integer constants $\Delta_1, \Delta_2 \geq 1$. It first partitions each relation into two parts, R^-, R^+ and S^-, S^+:

$$R^- = \{R(a,b) \mid |\sigma_{x=a}R(x,y)| \leq \Delta_2 \text{ or } |\sigma_{y=b}S(z,y)| \leq \Delta_1\}$$

$$S^- = \{S(c,b) \mid |\sigma_{z=c}S(z,y)| \leq \Delta_2 \text{ or } |\sigma_{y=b}S(z,y)| \leq \Delta_1\}$$

In other words, R^-, S^- include the tuples that contain at least one value with low degree. R^+, S^+ contain the remaining tuples from R, S respectively. Algorithm 1 describes the detailed steps for computing the join. It proceeds by performing a (disjoint) union of the following results:

(1) Compute $R^- \bowtie S$ and $R \bowtie S^-$ using any worst-case optimal join algorithm, then project.
(2) Materialize R^+, S^+ as two rectangular matrices and use matrix multiplication to compute their product.

Intuitively, the "light" values are handled by standard join techniques, since they will not result in a large intermediate result before the projection. On the other hand, since the "heavy" values will cause a large output, it is better to compute their result directly using (fast) matrix multiplication.

Analysis. We now provide a runtime analysis of the above algorithm, and discuss how to optimally choose Δ_1, Δ_2.

We first bound the running time of the first step. To compute the full join result (before projection), a worst-case optimal algorithm needs time $O(N_R + N_S + |\mathsf{OUT}_{\bowtie}|)$, where $|\mathsf{OUT}_{\bowtie}|$ is the size of the join. The main observation is that the size of the join is bounded by $N_S \cdot \Delta_1 + |\mathsf{OUT}| \cdot \Delta_2$. Hence, the running time of the first step is $O(N_R + N_S \cdot \Delta_1 + |\mathsf{OUT}| \cdot \Delta_2)$.

To bound the running time of the second step, we need to bound appropriately the dimensions of the two rectangular matrices that correspond to the subrelations R^+, S^+. Indeed, the heavy x-values for R^+ are at most N_R/Δ_2, while the heavy y-values are at most N_S/Δ_1. This is because $|\mathbf{dom}(y)| \leq N_S$. Hence, the dimensions of the matrix for R^+ are $(N_R/\Delta_2) \times (N_S/\Delta_1)$. Similarly, the dimensions of the matrix for S^+ are $(N_S/\Delta_1) \times (N_S/\Delta_2)$. The matrices are represented as two-dimensional arrays and can be constructed in time $C = \max\{N_R/\Delta_2 \cdot N_S/\Delta_1, N_S/\Delta_1 \cdot N_S/\Delta_2\}$ by simply iterating over all possible heavy pairs and checking whether they form a

tuple in the input relations. Thus, from Lemma 1 the running time of the matrix multiplication step is $M(\frac{N_R}{\Delta_2}, \frac{N_S}{\Delta_1}, \frac{N_S}{\Delta_2})$. Summing up the two steps, the cost of the algorithm is in the order of:

$$N_R + N_S \, \Delta_1 + |\mathsf{OUT}|\Delta_2 + M\left(\frac{N_R}{\Delta_2}, \frac{N_S}{\Delta_1}, \frac{N_S}{\Delta_2}\right) + C \quad (1)$$

Using the above formula, one can plug in the formula for the matrix multiplication cost and solve to find the optimal values for Δ_1, Δ_2. We show how to do this in Section 5.

In the next part, we provide a theoretical analysis for the case where matrix multiplication is achievable with the theoretically optimal $\omega = 2$ for the case where $N_R = N_S = N$. Observe that the matrix construction cost C is of the same order as $M(\frac{N_R}{\Delta_2}, \frac{N_S}{\Delta_1}, \frac{N_S}{\Delta_2})$ even when $\omega = 2$, since β is the smallest of the three terms $N_R/\Delta_2, N_S/\Delta_1, N_S/\Delta_2$. Thus, it is sufficient to minimize the expression

$$f(\Delta_1, \Delta_2) = N + N \cdot \Delta_1 + |\mathsf{OUT}| \cdot \Delta_2 + \frac{N^2}{\Delta_2 \min\{\Delta_1, \Delta_2\}}$$

while ensuring $1 \leq \Delta_1, \Delta_2 \leq N$.

The first observation is that for any feasible solution $f(x,y)$ where $x > y$, we can always improve the solution by decreasing the value of Δ_1 from x to y. Thus, w.l.o.g. we can impose the constraint $1 \leq \Delta_1 \leq \Delta_2 \leq N$.

Case 1. $|\mathsf{OUT}| \leq N$. Since $\Delta_1 \leq \Delta_2$, we have $f(\Delta_1, \Delta_2) = N \cdot \Delta_1 + |\mathsf{OUT}| \cdot \Delta_2 + N^2/\Delta_2 \cdot \Delta_1$. To minimize the running time we equate $\partial f/\partial \Delta_1 = N - N^2/(\Delta_2 \Delta_1^2) = 0$ and $\partial f/\partial \Delta_2 = \mathsf{OUT} - N^2/(\Delta_1 \Delta_2^2) = 0$. Solving this system of equations gives that the critical point has $\Delta_1 = |\mathsf{OUT}|^{1/3}$, $\Delta_2 = N/|\mathsf{OUT}|^{2/3}$. Since $|\mathsf{OUT}| \leq N$, this solution is feasible, and it can be verified that it is the minimizer of the running time, which becomes

$$N + N \cdot |\mathsf{OUT}|^{1/3}$$

Case 2. $|\mathsf{OUT}| > N$. For this case, there is no critical point inside the feasible region, so we will look for a minimizer at the border, where $\Delta_1 = \Delta_2 = \Delta$. This condition gives us $f(\Delta) = (N + |\mathsf{OUT}|) \cdot \Delta + N^2/\Delta^2$, with minimizer $\Delta = \left(2N^2/(N + |\mathsf{OUT}|)\right)^{1/3}$. The runtime then becomes

$$O(N^{2/3} \cdot |\mathsf{OUT}|^{2/3})$$

We can summarize the two cases with the following result.

Lemma 3. *Assuming that the exponent in matrix multiplication is $\omega = 2$, the query $\ddot{Q}$ can be computed in time*

$$O(|\mathbf{D}| + |\mathbf{D}|^{2/3} \cdot |\mathsf{OUT}|^{1/3} \cdot \max\{|\mathbf{D}|, |\mathsf{OUT}|\}^{1/3})$$

Lemma 2 implies a running time of $O(|\mathbf{D}| \cdot |\mathsf{OUT}|^{1/2})$ for $\ddot{Q}$, which is strictly worse compared to the running time of the above lemma *for every output size* $|\mathsf{OUT}|$.

Remark. For the currently best known value of $\omega = 2.37$, the running time is $O(|\mathbf{D}|^{0.83} \cdot |\mathsf{OUT}|^{0.589} + |\mathbf{D}| \cdot |\mathsf{OUT}|^{0.41})$. It

can also be shown that this algorithm is worst-case optimal (up to constant factors).

Comparing with previous results. We now discuss the result in [11], which uses matrix multiplication to give a running time of $\tilde{O}(|\mathbf{D}|^{0.862} \cdot |\text{OUT}|^{0.408} + |\mathbf{D}|^{2/3} \cdot |\text{OUT}|^{2/3})$. We point out an error in their analysis that renders their claim incorrect for the regime where $|\text{OUT}| < N$.

In order to obtain their result, the authors make a split of tuples into light and heavy, and obtain a formula for running time in the order of $N\Delta_b + |\text{OUT}|\Delta_{ac} + M\left(\frac{N}{\Delta_{ac}}, \frac{N}{\Delta_b}, \frac{N}{\Delta_{ac}}\right)$, where Δ_b, Δ_{ac} are suitable degree thresholds. Then, they use the formula from [26] for the cost of matrix multiplication, where $M(x, y, x) = x^{1.84} \cdot y^{0.533} + x^2$. However, this result can be applied only when $x \geq y$, while the authors apply it for regimes where $x < y$. (Indeed, if say $x = N^{0.3}$ and $y = N^{0.9}$, then we would have $M(x, y, x) = N^{1.03}$, which is smaller than the input size $N^{1.2}$.) Hence, the running time analysis is valid only when $N/\Delta_{ac} \geq N/\Delta_b$, or equivalently $\Delta_b \geq \Delta_c$. Since the thresholds are chosen such that $N\Delta_b = |\text{OUT}|\Delta_{ac}$, it means that the result is correct *only in the regime where* $|\text{OUT}| \geq N$. In other words, when the output size is smaller than the input size, the running time formula from [11] is not applicable.

In the case where $\omega = 2$, the cost formula from [26] becomes $M(x, y, x) = x^2$, and [11] gives an improved running time of $\tilde{O}(N^{2/3} \cdot |\text{OUT}|^{2/3})$. Again, this is applicable only when $|\text{OUT}| \geq N$, in which case it matches the bound from Lemma 3. Notice that for $|\text{OUT}| < N^{1/2}$ the formula would imply a deterministic sublinear time algorithm.

3.2 The Star Query

We now generalize the result to the star query $Q_k^\star$. As before, we assume that all tuples that do not contribute to the join output have already been removed.

Algorithm. The algorithm is parametrized by two integer constants $\Delta_1, \Delta_2 \geq 1$. We partition each relation R_i into three parts, R_i^+, R_i^- and $R_i^\diamond$:

$$R_i^- = \{R_i(a, b) \mid |\sigma_{x_i=a}R_i(x_i, y)| \leq \Delta_2\}$$
$$R_i^\diamond = \{R_i(a, b) \mid |\sigma_{y=b}R_j(x_j, y)| \leq \Delta_1, \text{ for each } j \in [k] \setminus i\}$$
$$R_i^+ = R_i \setminus (R_i^- \cup R_i^\diamond)$$

In other words, R_i^- contains all the tuples with light x, $R_i^\diamond$ contains all the tuples with y values that are light in all other relations, and R_i^+ the remaining tuples. The algorithm now proceeds by computing the following result:

(1) Compute $R_1 \bowtie \ldots R_j^- \bowtie \ldots R_k$ using any worst-case optimal join algorithm, then project for each $j \in [k]$.
(2) Compute $R_1 \bowtie \ldots R_j^\diamond \bowtie \ldots R_k$ using any worst-case optimal join algorithm, then project for each $j \in [k]$.

(3) Materialize $R_1^+, \ldots, R_k^+$ as rectangular matrices and use matrix multiplication to compute their product.

Analysis. We assume that all relation sizes are bounded by N. The running time of the first step is $O(|\text{OUT}| \cdot \Delta_2)$ since each light value of variable x_i in relation R_i contributes to at least one output result.

For the second step, the key observation is that since y is light in all the relations (except possibly R_i), the worst-case join size before projection is bounded by $O(N \cdot \Delta_1^{k-1})$, and hence the running time is also bounded by the same quantity.

The last step is more involved than simply running matrix multiplication. This is because for each output result formed by heavy x_i values in R_i^+ (say $t = (a_1, a_2, \ldots a_k)$), we need to count the number of y values that connect with each a_i in t. However, running matrix multiplication one at a time between two matrices only tells about the number of connection y values for any two pair of a_i and not all of t. In order to count the y values for all of t together, we divide variables $x_1, \ldots x_k$ into two groups of size $\lceil k/2 \rceil$ and $\lfloor k/2 \rfloor$ followed by creating two adjacency matrices. Matrix V is of size $\left(\frac{N}{\Delta_2}\right)^{\lceil k/2 \rceil} \times \frac{N}{\Delta_1}$ such that

$$V_{(a_1, a_2, \ldots a_{\lceil k/2 \rceil}), b} = \begin{cases} 1, & (a_1, b) \in R_1, \ldots, (a_{\lceil k/2 \rceil}, b) \in R_{\lceil k/2 \rceil} \\ 0, & \text{otherwise} \end{cases}$$

Similarly, matrix W is of size $\left(\frac{N}{\Delta_2}\right)^{\lfloor k/2 \rfloor} \times \frac{N}{\Delta_1}$ such that

$$W_{(a_{\lceil k/2 \rceil+1} \ldots a_k), b} = \begin{cases} 1, & (a_{\lceil k/2 \rceil+1}, b) \in R_{\lceil k/2 \rceil+1}, \ldots, (a_k, b) \in R_k \\ 0, & \text{otherwise} \end{cases}$$

Matrix construction takes time $(N/\Delta_2)^{\lceil k/2 \rceil} \cdot N/\Delta_1$ time in total. We have now reduced step three to computing the matrix product $V \times W^T$. Summing up the cost of all three steps, the total cost is as below which can be minimized similar to two-path query:

$$N \cdot \Delta_1^{k-1} + |\text{OUT}| \cdot \Delta_2 + M\left(\left(\frac{N}{\Delta_2}\right)^{\lceil k/2 \rceil}, \frac{N}{\Delta_1}, \left(\frac{N}{\Delta_2}\right)^{\lfloor k/2 \rfloor}\right)$$

3.3 Boolean Set Intersection

In this setting, we are presented with a workload W containing boolean set intersection (BSI) queries of the form $Q_{ab}() = R(a, y), S(b, y)$ parametrized by the constants a, b. The queries come at a rate of B queries per time unit. In order to service these requests, we can use multiple machines. Our goal is twofold: minimize the number of machines we use, while at the same time minimizing the *average latency*, defined as the average time to answer each query.

Our key observation is that, instead of servicing each request separately, we can *batch* the requests and compute them all at once. To see why this can be beneficial, suppose that we batch together C queries. Then, we can group all

Algorithm 2: `SizeAware` [21]

Input: Indexed sets $R = \{R_1, \dots, R_m\}$ and c
Output: Unordered SSJ result

1 *degree threshold* x = `GetSizeBoundary`(R, c), $\mathcal{L} \leftarrow \emptyset$
2 $R = R_l \cup R_h$ /* partition sets into light and heavy */
3 *Evaluate* $R \bowtie R_h$ *and enumerate result*
4 **foreach** $r \in R_l$ **do**
5   **foreach** c-*subset* r_c *of* r **do**
6        $\mathcal{L}[r_c] = \mathcal{L}[r_c] \cup r$
7 **foreach** $l \in \mathcal{L}[r_c]$ **do**
8   *enumerate every set pair in l if not output already*

---

pairs of constants (a, b) to a single binary relation $T(x, z)$ of size C, and compute the following conjunctive query:

$$Q_{batch}(x, z) = R(x, y), S(z, y), T(x, z).$$

Here, R, S have size N, and T has size C. The resulting output will give the subset of the pairs of sets that indeed intersect. The above query can be computed by applying a worst-case optimal algorithm and then performing the projection: this will take $O(N \cdot C^{1/2})$ time. Hence, the average latency for a request will be $\frac{C}{B} + \frac{N}{C^{1/2}}$. This idea can be extended to speed up the boolean version of the two-path query. We refer the reader to the full version of the paper for more details [19].

4 SPEEDING UP SSJ AND SCJ

In this section, we describe how to apply Algorithm 1 to speed up unordered SSJ. We refer the reader to the full version of this paper on how we can handle ordered SSJ and SCJ [19].

We first review the state-of-the-art algorithm from [21] called `SizeAware`. Algorithm 2 describes the size-aware set similarity join algorithm. The key insight is to identify a degree threshold x and partition the input sets into light and heavy. All heavy sets that form an output pair are enumerated by a sort-merge join. All light sets are processed by generating all possible c-sized subsets and then building an inverted index over it that allows for enumerating all light output pairs.x is chosen such that the cost of processing heavy and light sets is equal to each other.

We propose three key modifications that give us the new algorithm called `SizeAware++`. First, observe that $\mathcal{J}_H = R \bowtie R_h$ (line 3) is a natural join and requires $N \cdot N/x$ operations (recall that $|R_h| = N/x$ in the worst-case) even if the join output is smaller. Thus, Algorithm 1 is applicable here directly. This strictly improves the theoretical worst-case complexity of Algorithm 2 whenever $|\mathcal{J}_H| < N^2/x$ for all c.

The second modification is to deal with high duplication when enumerating all light pairs using the inverted index $\mathcal{L}[r_c]$. The key observation is that the algorithm in line 8

goes over all possible pairs and generates the full join result, which takes $|\mathcal{J}_L| = \sum_{r_c} |\mathcal{L}[r_c]|^2$ time. If the final output is smaller than $|\mathcal{J}_L|$, then the matrix multiplication-based algorithm can speed up the computation.

The third modification relates to optimizing the expansion of light nodes (line 3 in Algorithm 1). Recall that the algorithm expands all light y values. Suppose we have $R(x, y)$ and $S(z, y)$ relations indexed and sorted according to the variable order in the schema. Let $\mathcal{L}[b] = \{c \mid (c, b) \in S(z, y)\}$ denote the inverted index for relation S. The time required to perform the deduplication for a fixed value for x (say a) is $T = \sum_{b:(a,b)\in R} |\mathcal{L}[b]|$. This is unavoidable for overlap $c = 1$. However, it is possible that for $c > 1$, the output size is much smaller than T. In other words, the deduplication step is expensive when the overlap between $\mathcal{L}[b]$ is high for different values of b. The key idea is to reuse computation across multiple a if there is a shared prefix and high overlap as explained in the full version [19].

Next, we highlight the important aspects regarding the parallelization of SSJ. Partitioning the join $\mathcal{J}_H$ is straightforward since all parallel tasks require no synchronization and access the input data in a read-only manner. Parallelizing $\mathcal{J}_L$ in SizeAware is harder because each parallel task needs to coordinate in order to deduplicate multiple results across different c-subsets. On the other hand, using matrix multiplication allows for coordination-free parallelism as the matrix can be partitioned easily and each parallel task requires no interaction with each other.

5 COST-BASED OPTIMIZATION

In this section, we briefly describe some of our optimizations necessary to make our algorithms practical.

Estimating output size. So far, we have not discussed how to estimate $|\text{OUT}|$. We derive an estimate in the following manner. First, it is simple to show that the following holds for $\ddot{Q}$: $|\textbf{dom}(x)| \leq |\text{OUT}| \leq \min\{|\textbf{dom}(x)|^2, |\text{OUT}_{\bowtie}|\}$ and $|\text{OUT}_{\bowtie}| \leq N \cdot \sqrt{|\text{OUT}|}$. Thus, a reasonable estimate for $|\text{OUT}|$ is the geometric mean of $\max\{|\textbf{dom}(x)|, (|\text{OUT}_{\bowtie}|/N)^2\}$ and $\min\{|\textbf{dom}(x)|^2, |\text{OUT}_{\bowtie}|\}$. If $|\text{OUT}_{\bowtie}|$ is not *much larger* than N, then the full join size is also reasonable estimate.

Matrix multiplication cost. A key component of our techniques is matrix multiplication. Lemma 1 states the complexity of performing multiplication and also includes the cost of creating the matrices. However, in practice, this could be a significant overhead in terms of memory usage and time. Further, the scalability of matrix multiplication implementation itself is subject to the matrix size, the underlying linear algebra framework, and the hardware support (vectorization, SIMD instructions, multithreading support, etc.) In order

Algorithm 3: Cost Based Optimizer

Output: degree threshold Δ_1, Δ_2

1 *Estimate full join result* $|\text{OUT}_{\bowtie}|$ *and* $|\text{OUT}|$

2 **if** $|\text{OUT}_{\bowtie}| \leq 20 \cdot N$ **then**

3   *use worst-case optimal join algorithm*

4 $t_{light} \leftarrow |\text{OUT}_{\bowtie}|, t_{heavy} \leftarrow 0, \text{prev}_{light} \leftarrow \infty, \text{prev}_{heavy} \leftarrow 0, \Delta_1 = N$

5 **while** true **do**

6 $\text{prev}_{light} \leftarrow t_{light}, \text{prev}_{heavy} \leftarrow t_{heavy}$

7 $\text{prev}_{\Delta_1} \leftarrow \Delta_1, \text{prev}_{\Delta_2} \leftarrow \Delta_2$

8 $\Delta_1 \leftarrow (1 - \epsilon)\Delta_1, \Delta_2 \leftarrow N \cdot \Delta_1 / |\text{OUT}|$

9 $t_{light} \leftarrow T_I \cdot \text{sum}(y_{\Delta_1}) + \cdot T_I \cdot \text{sum}(x_{\Delta_2}) +$

10 $T_m \cdot |\textbf{dom}(x)| + T_s \cdot \text{cdfx}(y_{\Delta_1}) \cdot |\textbf{dom}(x)|$

11 $u, v, w \leftarrow \#\text{heavy } x, y, z \text{ values using count}(w_\delta)$

12 $t_{heavy} \leftarrow \hat{M}(u, v, w, co) + T_m \cdot (u \cdot v + u \cdot w)$

13   **if** $\text{prev}_{light} + \text{prev}_{heavy} \leq t_{light} + t_{heavy}$ **then**

14   **return** $\text{prev}_{\Delta_1}, \text{prev}_{\Delta_2}$

---

to minimize the runtime, we need to take into consideration all of the above system parameters to find the optimal thresholds.

Symbol	Description
T_s	avg time for sequential access
T_m	avg time for allocating 32 bytes of memory
co	number of cores available
$\hat{M}(u, v, w, co)$	estimate of time required to multiply matrices of dimension $u \times v$ and $v \times w$ using co cores
T_I	avg time for random access and insert

Table 1: Symbol definitions.

Algorithm 3 describes the cost-based optimizer used to find the best thresholds. To simplify the description, we describe the details for the case of $\ddot{Q}$ where $R = S$. If the full join result is not much larger than the size of input relation, then we can simply use any worst case optimal join algorithm. For our experiments, we set the upper bound for $|\text{OUT}_{\bowtie}|$ to be at most $20 \cdot N$. Beyond this point, we begin to see the benefit of using matrix multiplication for join-project computation.

To find the best possible estimates for Δ_1, Δ_2, we employ binary search over the value of Δ_2. In each iteration, we increase or decrease its value by a factor of $(1 - \epsilon)$ where ϵ is a constant [1]. Once we fix the value of Δ_1 and Δ_2, we can query our precomputed index structure to find the exact number of operations that will be performed for all light y values and all light x values. Then, we find the number of heavy remaining values and get the estimate for time required to compute the matrix product. We stop the process when the estimate

---

[1] We fix $\epsilon = 0.95$ for our experiments

of the total time exceeds that of the previous iteration. The entire process terminates in $O(\log^2 N)$ steps.

Note that since $\hat{M}(u, v, w, co)$ is system dependent, we precompute a table that stores the time required for different values of u, v, w, co. As a brute-force computation for all possible values is very expensive to store and compute, we store the time estimate for $\hat{M}(p, p, p, co)$ for $p \in \{1000, 2000, \ldots, 20000\}$, $co \in [5]$. Then, given an arbitrary u, v, w, co, we can extrapolate from the nearest estimate available from the table.

6 EXPERIMENTAL EVALUATION

In this section, we empirically evaluate the performance of our algorithms. The main goal of the section is four fold:

(1) Empirically verify the speed-up obtained for the 2-path and star queries using algorithm from Section 3 compared to Postgres, MySQL, EmptyHeaded [9] and Commercial database X.

(2) Evaluate the performance of our approach against SizeAware and SizeAware++ for unordered and ordered SSJ.

(3) Evaluate the performance our approach against three state-of-the-art algorithms, namely PIEJoin [29], LIMIT+ [15], and PRETTI for SCJ

(4) Validate batching for boolean set intersection.

All experiments are performed on a machine with Intel Xeon CPU E5-2660@2.6GHz, 20 cores and 150 GB RAM. Unless specified, all experiments are single threaded implementations. For all experiments, we focus on self-join i.e all relations are identical. All C++ code is compiled using clang 8.0 with `-Ofast` flag and all matrix multiplication related code is additionally compiled with `-mavx -mfma -fopenmp` flags for multi-core support. Each experiment is run 5 times and we report the running time by averaging three values after excluding the slowest and the fastest runtime.

6.1 Datasets

We conduct experiments on six real-world datasets from different domains. DBLP [1] is a bibliography dataset from DBLP representing authors and papers. RoadNet [5] is road network of Pennsylvania. Jokes [3] is a dataset scraped from Reddit where each set is a joke and there is an edge between a joke and a word if the work is present in the joke. Words [6] is a bipartite graph between documents and the lexical tokens present in them. Image [2] dataset is a graph where each image is connected to a feature attribute if the image contains the corresponding attribute and Protein [4] refers to a bipartite graph where an edge signifies interaction between two proteins. Table 2 shows the main characteristics of the datasets. DBLP and RoadNet are examples of sparse datasets whereas the other four are dense datasets.
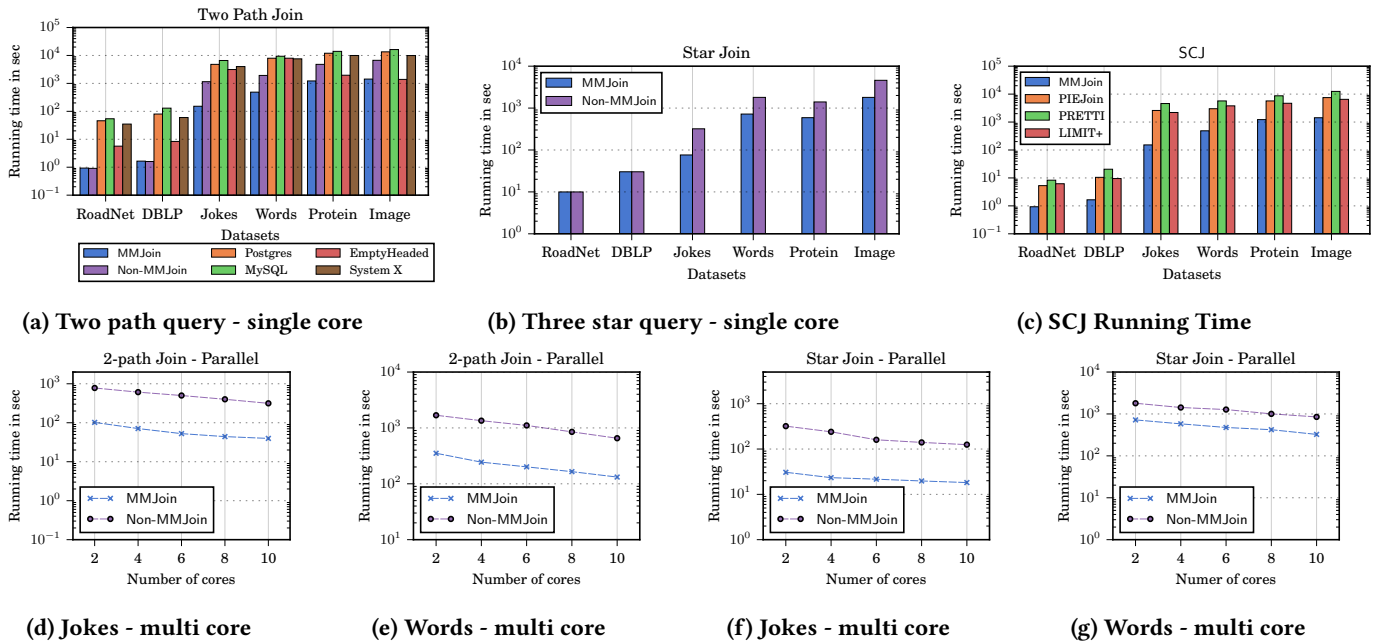
(a) Two path query - single core

(b) Three star query - single core

(c) SCJ Running Time

(d) Jokes - multi core

(e) Words - multi core

(f) Jokes - multi core

(g) Words - multi core

Figure 1: Join Processing for two path and star query

Dataset	$|R|$	No. of sets	$|\mathbf{dom}|$	Avg set size	Min set size	Max set size
DBLP	10M	1.5M	3M	6.6	1	500
RoadNet	1.5M	1M	1M	1.5	1	20
Jokes	400M	70K	50K	5.7K	130	10K
Words	500M	1M	150K	500	1	10K
Protein	900M	60K	60K	15K	50	50K
Image	800M	70K	50K	11.4K	10K	50K

Table 2: Dataset Characteristics

6.2 Join Processing

In this part, we evaluate the running time for two queries: $\ddot{Q}$ and $Q_3^\star$. To extract the maximum performance from Postgres, we use PGTune to set the best configuration parameters. This is important to ensure that the query plan does not perform nested loop joins. For all datasets, we build a hash index over each variable to ensure that the optimizer can choose the best query plan. We manually verify that the query plan generated by PostgreSQL (and MySQL) when running these queries chooses HashJoin or MergeJoin. For DBMS X, we allow up to 1TB of disk space and supply query hints to make sure that all resources are available for query execution. Figure 1a shows the run time for different algorithms on a single core. MySQL and Postgres have the slowest running time since they evaluate the full query join result and then deduplicate. DBMS X performs marginally better than MySQL and Postgres. The non-matrix multiplication join (denoted Non-MMJoin) based on Lemma 2 is the second-best algorithm. The matrix multiplication-based join (denoted MMJoin) is the fastest on all datasets except RoadNet and DBLP, where

the optimizer chooses to compute the full join. A reason for the huge performance difference between MMJoin and other algorithms is that deduplication by computing the full join result requires either sorting the data or using hash tables, both of which are expensive operations. By using matrix multiplication, we avoid materializing the large intermediate join result. EmptyHeaded performs comparably to MMJoin for the Jokes dataset and outperforms MMJoin slightly on the Image dataset. This is because the Image dataset contains a dense component. Since EmptyHeaded is designed as a linear algebra engine like Intel MKL, the performance is similar.

Figure 1d and Figure 1e show the performance of the combinatorial and non-combinatorial algorithm as the number of cores increases. Both algorithms demonstrate a speed-up. We omit MySQL and Postgres, since they do not allow for multi-core processing of single queries.

Next, we turn to the star query on three relations. For this experiment, we take the largest sample of each relation so that the result can fit in the main memory and the join can finish in reasonable time. Figure 1b shows the performance of the combinatorial and the non-combinatorial join on a single core. All other engines (except EmptyHeaded) failed to finish in 15000 seconds, except on RoadNet and DBLP. EmptyHeaded performs similarly to MMJoin on the Protein and Image datasets, but not on the other datasets. Figure 1f and Figure 1g show the performance in a multi-core setting.

(a) Image - single core   **(b) Jokes**   **(c) Words**   **(d) Image**

(e) DBLP - single core   **(f) Jokes - single core**   **(g) Image - single core**   **(h) DBLP - multi core**

(i) DBLP - single core   **(j) Jokes - single core**   **(k) Jokes - multi core**   **(l) Image - multi core**
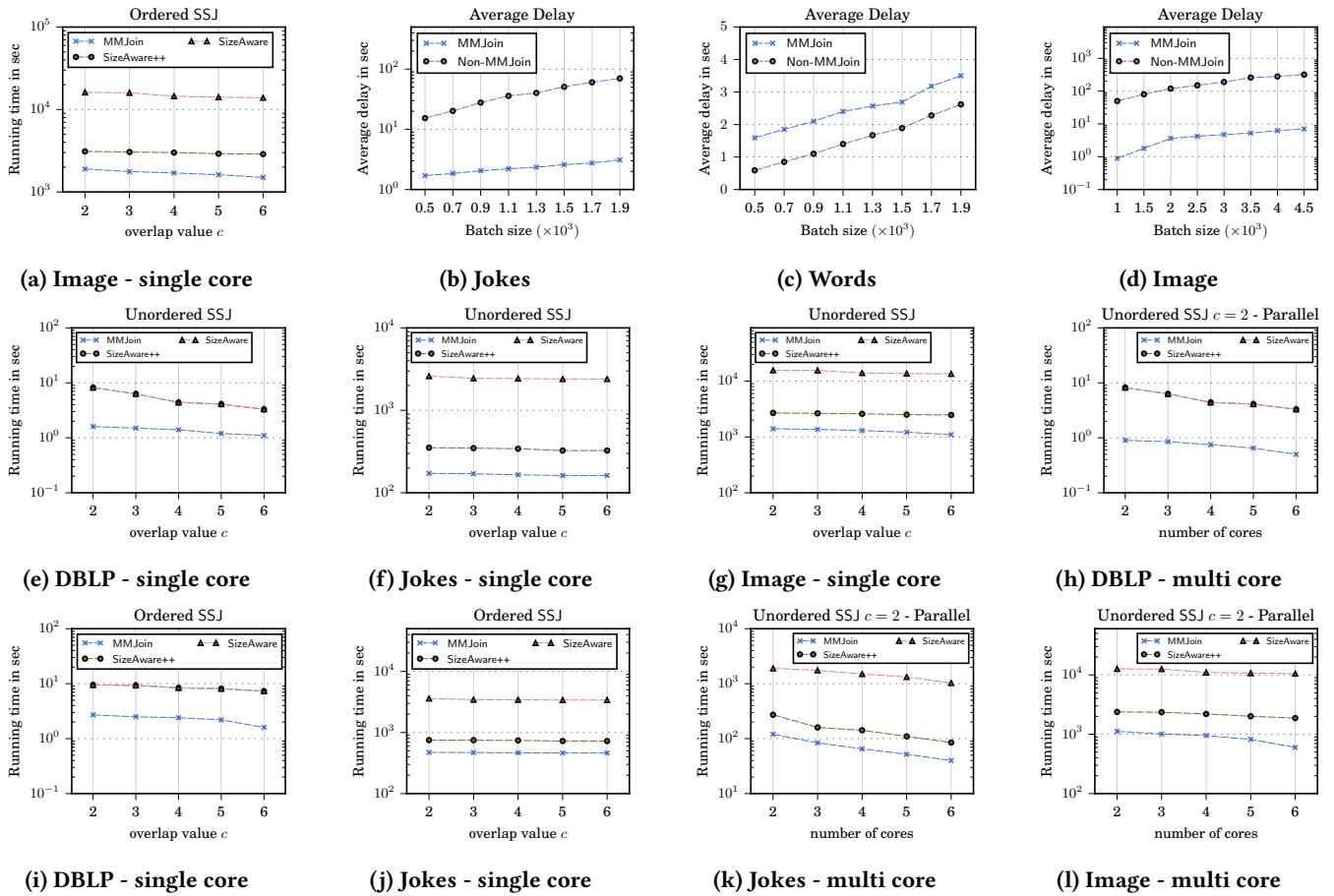
Figure 2: Ordered SSJ and minimizing average delay (row 1); Unordered and Ordered SSJ (row 2 and 3).

Once again, matrix multiplication performs better than its combinatorial counterpart across all experiments.

6.3 Set Similarity

In this section, we look at the set similarity join (SSJ). For both settings below, we switch on the prefix tree optimizations at all nodes.

Unordered SSJ. Figure 2e, Figure 2f and Figure 2g show the running time of MMJoin, SizeAware and SizeAware++ on a single core for the DBLP, Jokes, and Image datasets respectively. Since DBLP is a sparse dataset with small set sizes, MMJoin is the fastest and both SizeAware and SizeAware++ are marginally slower due to the optimizer cost. For Jokes and Image datasets, SizeAware is the slowest algorithm. This is because both the light and heavy processing have deduplication to perform. SizeAware++ is an order of magnitude faster than SizeAware, since it uses matrix multiplication but is slower than MMJoin because it still needs to enumerate the c-subsets before using matrix multiplication. MMJoin is

the fastest as it is output sensitive and performs the best in a setting with many duplicates. Next, we look at the parallel version of unordered SSJ. Figure 2h, Figure 2k and Figure 2l show the results for multi-core settings. For each experiment, we fix the overlap to $c = 2$. Both MMJoin and SizeAware++ are more scalable than SizeAware: this is because the light sets processing of SizeAware cannot be done in parallel while matrix multiplication-based deduplication can be performed in parallel.

Ordered SSJ. Recall that for ordered SSJ, our goal is to enumerate the set pairs in descending order of set similarity. Thus, once the set pairs and their overlap is known, we need to sort the result using overlap as the key. Figure 2i and Figure 2j show the running time for a single-threaded implementation. Compared to the unordered setting, the extra overhead of materializing the output and sorting the result increases the running time for all algorithms. For SizeAware, there is an additional overhead of finding the true overlap

for all light set pairs as well. Both MMJoin and SizeAware++ maintain their advantage similar to the unordered setting.

Impact of optimizations. Recall that SizeAware++ contains three optimizations: processing heavy sets using MMJoin, processing light sets via MMJoin and using prefix based materialization for computation sharing. Figure 3 shows the effect of switching on various optimizations. NO-OP denotes all optimizations switched off. The running time is shown as a percentage of the NO-OP running time (100%). Light denotes using two-path join on only light sets identified by SizeAware but not using the prefix optimizations. Heavy includes the Light optimizations switched on plus two-path join processing on the heavy sets. Finally, Prefix switches on materialization of the output in the prefix tree. Both Light and Heavy optimizations improve the running time by an order of magnitude, while Prefix further improves by a factor of 5×. We defer the experiments for SCJ to the full version of the paper [19].
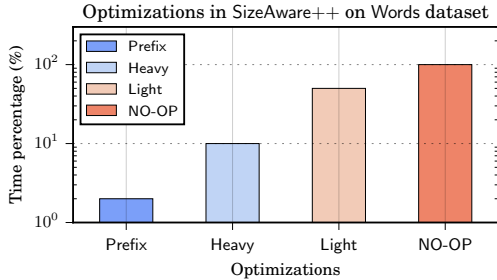

Figure 3: SSJ - Impact of optimizations on Words

6.4 Boolean Set Intersection

In this part of the experiment, we look at the boolean set intersection. The arrival rate of the queries is set to $B = 1000$ queries per second and our goal is to minimize the average delay metric as defined in Section 3.3. The workload is generated by sampling each set pair uniformly at random. We run this experiment for 300 seconds for each batch size and report the mean average delay metric value. Figure 2b, Figure 2c and Figure 2d show the average delay for the three datasets at different batch sizes. Recall that the smaller batch the size we choose, the larger is the number of processing units required. For the Jokes dataset, Non-MMJoin has the smallest average delay of $\approx 1s$ when $S = 10$. In that time, we collect a further 1000 requests, which means that there is a need for 100 parallel processing units. On the other hand, MMJoin achieves a delay of $\approx 2s$ at batch size 900. Thus, we need only 3 parallel processing units in total to keep up with the workload while paying only a small penalty in latency. For the Image dataset, MMJoin can achieve average delay of $1s$ at $S = 1000$ queries while Non-MMJoin achieves $50s$ at the same batch size. This shows that matrix multiplication

is useful for achieving a smaller latency using less resources, in line with the theoretical prediction. For the Words dataset, most of the sets have a small degree. Thus, the optimizer chooses to evaluate the join via the combinatorial algorithm.

7 RELATED WORK

Theoretically, [13] and [11] are the most closely related works to our considered setting (as discussed in Section 2.3). In practice, most of the previous work has considered join-project query evaluation by pushing down the projection operator in the query plan [14, 16, 23, 24]. LevelHeaded [8] and EmptyHeaded [9] are general linear algebra systems that use highly optimzed set intersections to speed up evaluation of cyclic joins, counting queries and support projections over them which is a more expressive set of queries compared to what our framework can handle. Since Intel MKL is also a linear algebra library, one can also use EmptyHeaded as the underlying framework for performing matrix multiplication. Very recently, [28] showed that for any hierarchical query (including star) with k relations, there exists an algorithm that pre-processes in time $T = O(N^{1+(k-1)\epsilon})$ such that it is possible to enumerate the join-project result without duplications with worst-case delay guarantee $\delta = O(N^{1+\epsilon})$ for any $\epsilon \in [0, 1]$, which implies a total running time of $\delta \cdot |\mathsf{OUT}|$. For group-by aggregate queries, [37] also used worst-case optimal join algorithms to avoid evaluating binary joins one at a time and materializing the intermediate results. However, the running time of their algorithm is not output-sensitive with respect to the final projected result and could potentially be improved upon by using our proposed ideas.

8 CONCLUSION AND FUTURE WORK

In the paper, we study the evaluation of join queries with projections. This is useful for a wide variety of tasks including set similarity, set containment and boolean query answering. We describe an algorithm based on fast matrix multiplication that allows for theoretical speedups. Empirically, we also demonstrate that the framework is practically useful and can provide speedups of up to 50× for some datasets. There are several promising future directions that remain to be explored. The first key direction is to extend our techniques to arbitrary acyclic queries with projections. Second, it remains unclear if the same techniques can also benefit cyclic queries or not. For example, the AYZ algorithm is applicable to counting cycles in graph using matrix multiplication. It would be interesting to extend the algorithm to enumerate the output of join-project queries when the user can choose arbitrary projection variables.

REFERENCES

[1] DBLP. https://dblp.uni-trier.de/.

[2] image. https://cs.stanford.edu/~acoates/stl10/.

[3] Jokes. https://goldberg.berkeley.edu/jester-data/.

[4] protein. https://string-db.org/cgi/download.pl?sessionId=IBdaKPtZGbl2.

[5] RoadNet. https://snap.stanford.edu/data/roadNet-PA.html.

[6] words. https://archive.ics.uci.edu/ml/datasets/bag+of+words.

[7] I. Abdelaziz, R. Harbi, S. Salihoglu, P. Kalnis, and N. Mamoulis. Spartex: A vertex-centric framework for rdf data analytics. *Proceedings of the VLDB Endowment*, 8(12):1880–1883, 2015.

[8] C. Aberger, A. Lamb, K. Olukotun, and C. Ré. Levelheaded: A unified engine for business intelligence and linear algebra querying. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 449–460. IEEE, 2018.

[9] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)*, 42(4):20, 2017.

[10] P. Afshani and J. A. S. Nielsen. Data structure lower bounds for document indexing problems. In *ICALP 2016Automata, Languages and Programming*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik GmbH, 2016.

[11] R. R. Amossen and R. Pagh. Faster join-projects and sparse matrix multiplications. In *Proceedings of the 12th International Conference on Database Theory*, pages 121–126. ACM, 2009.

[12] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. *SIAM Journal on Computing*, 42(4):1737–1767, 2013.

[13] G. Bagan, A. Durand, and E. Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, pages 208–222, 2007.

[14] G. Bhargava, P. Goel, and B. R. Iyer. Enumerating projections in sql queries containing outer and full outer joins in the presence of inner joins, Nov. 11 1997. US Patent 5,687,362.

[15] P. Bouros, N. Mamoulis, S. Ge, and M. Terrovitis. Set containment join revisited. *Knowledge and Information Systems*, 49(1):375–402, 2016.

[16] S. Ceri and G. Gottlob. Translating sql into relational algebra: Optimization, semantics, and equivalence of sql queries. *IEEE Transactions on software engineering*, (4):324–345, 1985.

[17] H. Cohen and E. Porat. Fast set intersection and two-patterns matching. *Theoretical Computer Science*, 411(40-42):3795–3800, 2010.

[18] H. Cohen and E. Porat. On the hardness of distance oracle for sparse graph. *arXiv preprint arXiv:1006.1117*, 2010.

[19] S. Deep, X. Hu, and P. Koutris. Fast join project query evaluation using matrix multiplication. https://arxiv.org/abs/2002.12459.

[20] S. Deep and P. Koutris. Compressed representations of conjunctive query results. *arXiv preprint arXiv:1709.06186*, 2017.

[21] D. Deng, Y. Tao, and G. Li. Overlap set similarity joins with theoretical guarantees. In *Proceedings of the 2018 International Conference on Management of Data*, pages 905–920. ACM, 2018.

[22] F. L. Gall and F. Urrutia. Improved rectangular matrix multiplication using powers of the coppersmith-winograd tensor. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1029–1046. SIAM, 2018.

[23] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. 1995.

[24] A. Gupta, V. Harinarayan, and D. Quass. Generalized projections: a powerful approach to aggregation. Technical report, Stanford InfoLab, 1995.

[25] J. E. Hopcroft, J. D. Ullman, and A. Aho. The design and analysis of computer algorithms, 1975.

[26] X. Huang and V. Y. Pan. Fast rectangular matrix multiplication and applications. *Journal of complexity*, 14(2):257–299, 1998.

[27] R. Jampani and V. Pudi. Using prefix-trees for efficiently computing set joins. In *International Conference on Database Systems for Advanced Applications*, pages 761–772. Springer, 2005.

[28] A. Kara, M. Nikolic, D. Olteanu, and H. Zhang. Trade-offs in static and dynamic evaluation of hierarchical queries. *arXiv preprint arXiv:1907.01988*, 2019.

[29] A. Kunkel, A. Rheinländer, C. Schiefer, S. Helmer, P. Bouros, and U. Leser. Piejoin: towards parallel set containment joins. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management*, page 11. ACM, 2016.

[30] Y. Luo, G. H. Fletcher, J. Hidders, and P. De Bra. Efficient and scalable trie-based algorithms for computing set containment relations. In *2015 IEEE 31st International Conference on Data Engineering*, pages 303–314. IEEE, 2015.

[31] S. Misra, R. Barthwal, and M. S. Obaidat. Community detection in an integrated internet of things and social network architecture. In *2012 IEEE Global Communications Conference (GLOBECOM)*, pages 1647–1652. IEEE, 2012.

[32] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, pages 37–48. ACM, 2012.

[33] H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Record*, 42(4):5–16, 2013.

[34] M. Patrascu and L. Roditty. Distance oracles beyond the thorup-zwick bound. In *Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on*, pages 815–823. IEEE, 2010.

[35] T. Veldhuizen. Leapfrog triejoin: a worst-case optimal join algorithm, icdt,(2014).

[36] K. Xirogiannopoulos and A. Deshpande. Extracting and analyzing hidden graphs from relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 897–912. ACM, 2017.

[37] K. Xirogiannopoulos and A. Deshpande. Memory-efficient group-by aggregates over multi-way joins. *arXiv preprint arXiv:1906.05745*, 2019.

[38] K. Xirogiannopoulos, U. Khurana, and A. Deshpande. Graphgen: Exploring interesting graphs in relational data. *Proceedings of the VLDB Endowment*, 8(12):2032–2035, 2015.

[39] K. Xirogiannopoulos, V. Srinivas, and A. Deshpande. Graphgen: Adaptive graph processing using relational databases. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*, GRADES'17, pages 9:1–9:7, New York, NY, USA, 2017. ACM.

[40] J. Yang, W. Zhang, S. Yang, Y. Zhang, X. Lin, and L. Yuan. Efficient set containment join. *The VLDB Journal—The International Journal on Very Large Data Bases*, 27(4):471–495, 2018.