Density-optimized Intersection-free Mapping and Matrix Multiplication for Join-Project Operations

Zichun Huang
SKLP, Institute of Computing Technology, CAS
huangzichun21s@ict.ac.cn

Shimin Chen*
SKLP, Institute of Computing Technology, CAS
chensm@ict.ac.cn

ABSTRACT

A Join-Project operation is a join operation followed by a duplicate eliminating projection operation. It is used in a large variety of applications, including entity matching, set analytics, and graph analytics. Previous work proposes a hybrid design that exploits the classical solution (i.e., join and deduplication), and MM (matrix multiplication) to process the sparse and the dense portions of the input data, respectively. However, we observe three problems in the state-of-the-art solution: 1) The outputs of the sparse and dense portions overlap, requiring an extra deduplication step; 2) Its table-to-matrix transformation makes an over-simplified assumption of the attribute values; and 3) There is a mismatch between the employed MM in BLAS packages and the characteristics of the Join-Project operation.

In this paper, we propose DIM^3, an optimized algorithm for the Join-Project operation. To address 1), we propose an intersection-free partition method to completely remove the final deduplication step. For 2), we develop an optimized design for mapping attribute values to natural numbers. For 3), we propose DenseEC and SparseBMM algorithms to exploit the structure of Join-Project for better efficiency. Moreover, we extend DIM^3 to consider partial result caching and support Join-*op* queries, including Join-Aggregate and MJP (Multi-way Joins with Projection). Experimental results using both real-world and synthetic data sets show that DIM^3 outperforms previous Join-Project solutions by a factor of 2.3×-18×. Compared to RDBMSs, DIM^3 achieves orders of magnitude speedups.

PVLDB Reference Format:
Zichun Huang and Shimin Chen. Density-optimized Intersection-free Mapping and Matrix Multiplication for Join-Project Operations. PVLDB, 15(10): 2244 - 2256, 2022.
doi:10.14778/3547305.3547326

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/schencoding/JoinProject-DIM3.

1 INTRODUCTION

A Join-Project operation is a join operation followed by a duplicate eliminating projection operation [1]. Given two tables $R(x, y)$ and $S(z, y)$, the Join-Project operation can be written as follows:

$$\Pi_{x,z}(R(x,y) \bowtie_y S(z,y)) \quad (1)$$

It joins R and S with y as the join key, then projects and deduplicates (x,z) tuples. Π denotes the duplicate eliminating projection.

Example: In the HetRec2011 data set [7], *U(userID, bookID, tagID)* table contains tags given by users to books that they read, and *B(bookID, tagID, weight)* table records books and their possible tags with weights. The following SQL query recommends books to users based on the tags recorded in users' reading history:

SELECT DISTINCT U.userID, B.bookID
FROM U, B
WHERE U.tagID = B.tagID

This Join-Project operation joins U and B on tagID, then projects and deduplicates using SELECT DISTINCT. The results can be stored by the application for quick user-specific recommendations.

The Join-Project operation is used in a large variety of applications [10], including entity matching, set analytics, and graph analytics. The above is an example of entity matching. Similar examples include finding users who have seen the same movies in the MovieLens data set [15], and discovering co-authors in the DBLP data set [44]. Moreover, if tuple (x,y) represents that set x contains element y, then the Join-Project operation using y as the join key obtains all the pairs of sets that intersect with each other. Furthermore, if we interpret tuple (x,y) as an edge between two vertices x and y in a graph, then the Join-Project operation can be used to compute all pairs of vertices that are indirectly connected.

1.1 Previous Solutions

Classical Solution. The classical solution to compute the Join-Project in RDBMSs is to first perform the join operation [2, 24, 34], then deduplicate the projected join results using hash tables [25] or other types of indices [14, 17, 26]. The time complexity of the classical solution is $\Theta(|R|+|S|+|OUT_J|)$, where $|R|$, $|S|$, and $|OUT_J|$ denote the sizes of input table R, table S, and the join results before deduplication, respectively. This cost is reasonable when the number of duplicates is low. However, the solution is less efficient when $|OUT_J|$ is much larger than the size $|OUT_P|$ of the final results after deduplication. For example, $|OUT_J|$ is 3.7x as large as $|OUT_P|$ in the HetRec2011 example, while $|OUT_J|$ is 24x larger than $|OUT_P|$ in the MovieLens data set. Let $|X|$, $|Y|$ and $|Z|$ denote the number of distinct values in column x, y, and z in Eqn 1, respectively. Consider the case where $|X|$=$|Y|$=$|Z|$=n. $|OUT_J|$ can be $O(n^3)$ in the worst case, while $|OUT_P|$ is only $O(n^2)$. In other words, the classical solution can spend a lot of time generating the $|OUT_J|$ join results and then processing them to remove a large number of duplicates.

*Shimin Chen is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 15, No. 10 ISSN 2150-8097.
doi:10.14778/3547305.3547326

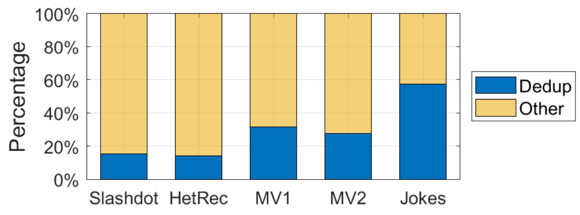


Figure 1: Breakdown of DHK run time.

Matrix Multiplication. Alternatively, the Join-Project operation can be computed using MM. The basic idea is to represent tables $R(x, y)$ and $S(z, y)$ as two matrices $\mathrm{R}^{x\times y}$ and $\mathrm{S}^{y\times z}$. Specifically, $\mathrm{R}_{x_i,y_k} = 1$ if and only if tuple $(x_i,y_k) \in R$ (similarly for S). Then the multiplication of $\mathrm{R}^{x\times y}$ and $\mathrm{S}^{y\times z}$ gives matrix $\mathrm{C}^{x\times z}$, where $\mathrm{C}_{x_i,z_j} = \sum_{k=1}^{|Y|} \mathrm{R}_{x_i,y_k}\mathrm{S}_{y_k,z_j}$. A non-zero element $\mathrm{C}_{x_i,z_j} > 0$ in the matrix corresponds to a tuple (x_i,z_j) in the final output of the Join-Project operation. Compared to the classical solution, MM performs the join and the deduplication together. There are efficient MM implementations in BLAS (Basic Linear Algebra Subprograms) packages with advanced techniques [9, 27, 38]. Moreover, there are sub-cubic MM algorithms in theory. The best known is the Coppersmith-Winograd algorithm with $O(n^{2.373})$ complexity [12].

Hybrid Solution. Recent studies [1, 10] combine the classical solution and the MM solution based on the observation that the classical solution performs better when the data is sparse, while MM performs better when the data is dense. Amossen et al. [1] propose an algorithm to partition the data into dense and sparse parts according to degrees of x, y, and z. Then dense MM is employed for the dense parts and the classical solution is used for the remaining parts.

Deep, Hu, and Koutris [10] correct errors in the cost analysis of [1] and implement the algorithm for experimental comparison. We call this state-of-the-art algorithm *DHK*. DHK has been shown to significantly outperform the classical solution. However, the following problems reduce the efficiency and practicality of DHK:

- *Overlapping outputs between sparse and dense parts*: The Join-Project results computed from the sparse and from the dense parts can overlap. Consequently, a final deduplication step is necessary to deduplicate the overlapping results. Figure 1 shows the breakdown of the run time of DHK for five data sets (cf. Section 5). For all these data sets, DHK partitions the input data into dense and sparse parts. It performs the final deduplication step, incurring significant overhead.
- *Over-simplified assumption for table-to-matrix transformation*: The DHK implementation assumes that the input columns x, y, and z contain consecutive natural numbers starting from 0. Thus, it directly uses their values as row or column ids in the matrices. However, in reality, database attribute values are rarely natural numbers. A step is missing: mapping values of input columns to consecutive natural numbers.
- *Caveats of MM Implementation*: DHK invokes MM in BLAS packages (e.g., Intel MKL [43]) as a black box. On the one hand, BLAS packages typically implement the $O(n^3)$ MM algorithm. It would be interesting to investigate sub-cubic MM algorithms. On the other hand, the MM invocation as a black box cannot exploit the characteristics of the Join-Project operations for performance improvement.

1.2 Our Solution: DIM^3

In this paper, we propose an efficient and practical Join-Project algorithm, DIM^3 (Density-optimized Intersection-free Mapping and Matrix Multiplication). We address the above problems as follows:

- *Intersection-free partitioning*: We propose a novel partitioning method that divides matrix $\mathrm{S}^{y\times z}$ into subsets of rows based on the density of z_j rows and then chooses different evaluation strategies for the dense rows and the sparse rows. Since the results (x_i,z_j) of the two parts have different z_js, this method is guaranteed to be intersection-free. Hence, DIM^3 completely removes the final deduplication step required by DHK.
- *Optimized mapping design*: We investigate the design of the mapping step in DIM^3. First, we identify cases (e.g., auto increment, dictionary encoding) where columns contain roughly natural numbers and thus the mapping step can be skipped. Second, we exploit the mapping of the shared join key columns to perform a semi-join like optimization to discard tuples without matches. Finally, we design a cache-optimized hash-based algorithm to efficiently compute the mappings.
- *Optimized MM algorithms*: We obtain and evaluate an implementation of the sub-cubic Strassen algorithm [41] in a recent study [3]. However, our results show that it cannot beat Intel MKL (cf. Section 5.2). Therefore, we focus on $O(n^3)$ algorithms in DIM^3. For the dense rows, we observe that the computation $\mathrm{C}_{x_i,z_j} = \sum_{k=1}^{|Y|} \mathrm{R}_{x_i,y_k}\mathrm{S}_{y_k,z_j}$ can stop early as soon as there is a non-zero $\mathrm{R}_{x_i,y_k}\mathrm{S}_{y_k,z_j}$. We design the *DenseEC* (Dense MM with Early stopping Checking) algorithm. For the sparse rows, we design the *SparseBMM* (Sparse Boolean MM) algorithm that leverages the CSR (Compressed Sparse Row) [23] format of matrix $\mathrm{S}^{y\times z}$ as a hash table on the join key y with NO hash conflicts. We also introduce a way to reduce the cost for initializing the deduplication vector.

In addition to addressing the three problems, we extend the Join-Project solution in the following two directions:

- *Partial Result Caching*: We observe that in real-world data sets, computing results for different z_j values often take widely different amounts of time. This motivates us to investigate if caching results for a subset of z_j values is profitable. We consider caching either the original or the complement set of x's given a z_j. The caching decision for z_j is based on a score computed from the computation time and the required result space. Our experiments show that partial result caching can effectively reduce the Join-Project computation time with reasonable space cost for most data sets.
- *Support for Join-Op Queries*: We discuss different types of relational operations op, and investigate whether we can leverage the Join-Project algorithm to support Join-op queries. We study two interesting ops in depth: 1) For Join-Aggregate queries [18], we show that DIM^3 can be applied with simple modifications; and 2) For MJP (Multi-Way Joins with Projection) queries, a.k.a. conjunctive queries with projection [11], we develop a dynamic programming algorithm to find the optimal query plan that considers pushing down deduplication operations to after the join operations.

1.3 Contributions

The main contributions of this paper is threefold. First, we propose DIM^3 with intersection-free partitioning, optimized mapping, and DenseEC and SparseBMM algorithms to address the three problems in the state-of-the-art solution (cf. Section 2). Second, we propose partial result caching for the Join-Project algorithm (cf. Section 3) and generalize DIM^3 to support Join-*op* queries (cf. Section 4). Third, we perform extensive experimental evaluation using both real-world and synthetic data sets (cf. Section 5). Experimental results show that DIM^3 outperforms previous Join-Project solutions by a factor of 2.3×-18×. Compared to commercial and open-source RDBMSs, DIM^3 achieves orders of magnitude speedups.

2 DIM^3 FOR JOIN-PROJECT

In this section, we first overview the DIM^3 algorithm in Section 2.1. Then, we explain the components of the algorithm in detail. Specifically, we present the mapping phase in Section 2.2, intersection-free partitioning in Section 2.3, SparseBMM in Section 2.4, and DenseEC in Section 2.5. Finally, we analyze the algorithm and describe the strategy selection criteria in Section 2.6. (An extended version of the paper provides time complexity analysis for our solutions [20].)

2.1 Overview

The DIM^3 algorithm is depicted in Figure 2 and listed in Algorithm 1. We perform the Join-Project operation on two tables $R(x, y)$ and $S(z, y)$. Without loss of generality, suppose R is the larger table and S is the smaller table.

DIM^3 begins by selecting either the classical or the hybrid solution for evaluating the Join-Project operation. This allows completely avoiding the mapping step, which can have significant cost for highly sparse data. DHK [10] uses a rule-of-thumb condition to choose the classical solution. In comparison, our strategy selection function f_1 makes better decisions based on the estimated run times of the classical and the hybrid solutions (cf. Section 2.6 and 5.2).

In the hybrid strategy, the first step is to map columns x, y, and z to consecutive natural numbers. In this way, an (x, y) tuple in R (similarly (y, z) tuple in S) can be converted to an element at row x and column y in matrix $\mathbf{R}^{x \times y}$. DHK [10] makes the over-simplified assumption that the input columns contain consecutive natural numbers. We delve into the design of the mapping step to provide general-purpose support for all attribute types.

The second step converts table R to CSR [23] format, then partitions S to S_{sparse} and S_{dense}. We propose an intersection-free partition method so that the Join-Project results of S_{sparse} and S_{dense} can be simply combined without the final deduplication step required by DHK [10]. The partition method uses function f_2 to decide which z row is dense. f_2 will be described in Section 2.6.

The third step designs SparseBMM and DenseEC algorithms for processing the sparse and dense data, respectively. SparseBMM computes the Join-Project $Result_{sparse}$ on R_{CSR} and S_{sparse}. DenseEC multiplies R_{CSR} with S_{dense} to obtain $Result_{dense}$. In DenseEC, function f_3 is used to choose the best method for intersecting two bitmaps. The computation of f_3 will be discussed in Section 2.6.

The final step is to merge $Result_{MM}$ and $Result_{EC}$ to obtain the final results. Because of intersection-free partitioning, there is no need to perform an extra deduplication step.

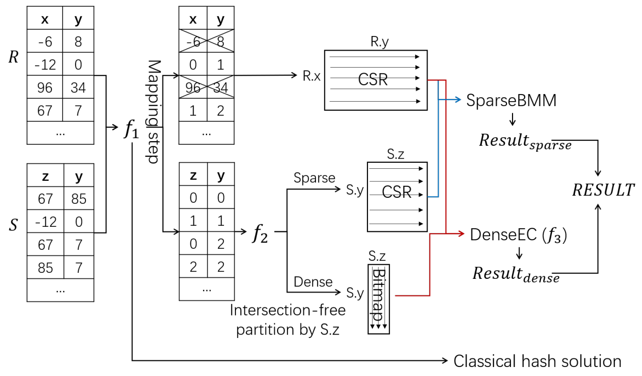


Figure 2: DIM^3 for Join-Project.

Algorithm 1: DIM^3

```
  Input: Table $R(x, y)$ and Table $S(z, y)$
  Output: List of result tuples $(x, z)$
1 Estimate $OUT_J$;
2 if $f_1(|R|, |S|, |OUT_J|) > 0$ then /* Classical is better */
3     return Use classical solution;
4 Mapping column values to consecutive natural numbers;
5 $R_{CSR} \leftarrow$ Create CSR for $R(x, y)$;
6 Intersection-free partition $S$ by $S.z$; /* dense if $f_2(z)>0$ */
7 $S_{sparse} \leftarrow$ Save the sparse part of $S$ as CSR;
8 $S_{dense} \leftarrow$ Save the dense part of $S$ as Bitmap array;
9 $Result_{sparse} \leftarrow$ SparseBMM($R_{CSR}$, $S_{sparse}$);
10 $Result_{dense} \leftarrow$ DenseEC($R_{CSR}$, $S_{dense}$);
11 return $Result_{sparse} \cup Result_{dense}$;
```

2.2 Mapping

The mapping step maps columns x, y, and z to consecutive natural numbers. This can be achieved with a baseline hash-based algorithm. Given a column, the algorithm looks up the values of the column one by one in a hash table. If a value v does not exist in the hash table, it inserts (v, the next consecutive number) into the hash table. As a result, every distinct value in the column is assigned a natural number. We can repeat this algorithm for x, y, and z.

This mapping algorithm can be costly. It creates three hash tables for x, y, and z, and performs up to $2(|R| + |S|)$ hash table lookups and/or inserts. Compared to the hash join of $R(x, y)$ and $S(z, y)$, which performs $|R| + |S|$ hash table accesses, the mapping algorithm pays twice as much cost for hash table visits. This can be a significant additional overhead for the Join-Project algorithm when the join result size is not much larger than the input sizes.

In the following, we consider three opportunities to optimize the baseline algorithm. Then, we extend the mapping to support a wider range of Join-Project operations.

Optimization 1: Skip Mapping. First of all, DIM^3 chooses the classical solution for highly sparse data sets as shown in Figure 2. This completely avoids the mapping step. Second, it is possible to skip mapping for columns that already contain natural numbers. There are two common cases in database systems. (i) Columns declared with auto increment (e.g., *AUTO_INCREMENT* in Oracle, *IDENTITY* in SQLServer and DB2, *SERIAL* in PostgreSQL, *AUTOINCREMENT* in MySQL) contain consecutive natural numbers. (ii) String columns

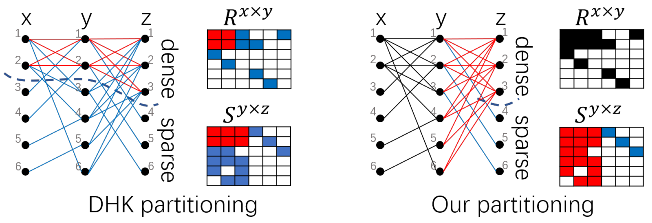


Figure 3: Comparing the partition methods in DHK [10] and DIM^3. (A tuple in a table is displayed as an edge in graphs and an element in matrices. Red: dense, blue: sparse)

can be encoded by dictionary encoding [22] and stored as natural numbers in database systems (e.g., SAP HANA and MonetDB).

Optimization 2: Reduce Computation with Join Key Mapping. The naïve way to map y is to use $R.y \cup S.y$ as the mapping input. We observe that only $R.y \cap S.y$ contributes to the equality join results. Therefore, we can employ a semi-join like idea, and optimize the mapping of y as follows. Since S is the smaller table, we first compute the mapping of $S.y$ using a hash table. Then we map tuples in R using the same hash table. If $y_k \in R.y$ but $y_k \notin S.y$, then the corresponding R tuple can be safely discarded because it does not have any matches in S. Note that we choose not to pay the cost of re-scanning S to remove S tuples with $y_k \in S.y$ but $y_k \notin R.y$. As shown in Figure 2, the first and third R tuples are filtered out.

Optimization 3: Optimize Hash Table Performance. When a hash table is larger than the CPU cache, hash table accesses result in expensive random memory accesses with poor CPU cache behavior. A hash table visit may probe multiple locations, and dereference pointers (e.g., in the case of chained hash table), incurring significant overhead. Therefore, we employ the following designs to improve the hash table performance in the mapping algorithm. First, we estimate the hash table size for a column (e.g., based on statistics of the number of distinct keys). If the size exceeds the last-level CPU cache, we employ cache partitioning. We use the last k bits of the hash value to divide the data into 2^k partitions so that the hash table of each partition fits into the last-level CPU cache. Then, we compute the mapping for each partition. Second, we employ a linear probing hash table design to avoid pointer dereference. We tune the number of slots and the maximum probing distance to reduce the cost of hash table accesses. If no available slots are found for a given column value, we employ a stash hash table (Flat_hash_map [40] in our implementation) to store the overflow data.

Supporting Wider Range of Join-Project Operations. We can map not only single attribute but also multiple attributes to consecutive numbers. For instance,

$$\Pi_{a,b,c,d,e}(R(a,b,c,d) \bowtie_{c,d} S(c,d,e,f))$$

can be treated as

$$\Pi_{x,z}(R(x,y) \bowtie_{y} S(z,y))$$

where $R.x = \{a, b\}$, $S.z = \{c, d, e\}$, and $R.y = S.y = \{c, d\}$. In this way, we can support any combinations of join keys and projection columns, including operations that contain the join key in the output.

2.3 Intersection-Free Partitioning

Figure 3 compares the partition methods of DHK [10] and DIM^3. DHK makes separate decisions on x, y and z according to their degrees (i.e. the number of tuples of the same attribute value). An (x, y) tuple is added to the dense part only if both its x and y attributes are considered as dense (shown as red color in the figure).

In comparison, we propose an intersection-free partition method[1] as shown in Figure 3. It examines table S and ensure that all tuples with the same $S.z$ value are in the same partition. Note that table R will not be partitioned. Essentially, we partition matrix S by its columns. Our partitioning method has the following benefits.

First, the Join-Project results generated by the sparse and dense parts do not intersect. Given a result (x,z), if z is judged as dense, this result must be generated by the dense part. Otherwise, it is generated by the sparse part. In contrast, this property does not hold in DHK. As shown in Figure 3, the result (x_2, z_2) is generated both in the dense part by joining (x_2, y_1) and (y_1, z_2), and in the sparse part by joining (x_2, y_3) and (y_3, z_2). Therefore, while DHK must deduplicate the results from the two parts, DIM^3 can eliminate this final deduplication step.

Second, DIM^3 may apply dense MM to more tuples. DHK considers a tuple as dense only if both its attributes are judged as dense. In comparison, DIM^3 makes the partitioning decision based solely on $S.z$. Since its dense criteria tend to be more flexible, DIM^3 can employ dense MM under more scenarios, as illustrated in Figure 3.

Third, the selection based on z simplifies the Join-Project computation. In DHK, it is costly (spatially) to record the per-tuple density decisions. Therefore, DHK does not save them. Instead, when processing the sparse part, DHK uses the degree thresholds to re-compute whether a tuple is dense and should be skipped. In comparison, DIM^3 avoids this complexity. As shown in Figure 2, table S is divided into the sparse and the dense parts based on $S.z$. Hence, there is no need to re-evaluate the density criteria any more.

Finally, the cost for computing the partition decision in DIM^3 is lower compared to DHK. DHK makes $|R| + |S|$ decisions on all input tuples. In comparison, DIM^3 makes $|Z|$ decisions on $S.z$. The number of decisions to make is much smaller. Consider the case where $|X|=|Y|=|Z|=n$. In the worst case, the cost is $O(n^2)$ in DHK, but only $O(n)$ in DIM^3. Moreover, DHK performs binary search to determine the density threshold, which incurs additional cost.

2.4 SparseBMM

The classical hash-based solution is often used to process the sparse part of the data. We propose a *SparseBMM* algorithm with two main optimization techniques, as shown in Algorithm 2.

First, we observe that hash table accesses are often one main cost of the classical hash-based computation. Interestingly, since the column values are mapped to natural numbers, the CSR (Compressed Sparse Row) [23] format of matrix $S^{y \times z}$ is essentially a hash table on the join key y with NO hash conflicts. The original CSR structure consists of three arrays: $Val[]$, $Col[]$, and $RowPtr[]$. $Val[]$ and $Col[]$ contain the value and the column index of non-zero elements in the matrix, respectively. $RowPtr[]$ points to the row starts in $Val[]$ and $Col[]$. In the case of Join-Project, $Val[]$ contains all 1's and can be omitted. Therefore, we have two arrays $Col[]$ and $RowPtr[]$. We employ matrix $S^{y \times z}$ as the hash table. Given y_k, we locate

[1]Previous work proposes an intersection-free partition method in the context of query enumeration algorithms [11]. It divides the data into two sets with roughly equal number of join results. The method optimizes enumeration delays rather than end-to-end query run times. It is not directly applicable to the Join-Project operation.

Algorithm 2: SparseBMM (for sparse data).

```
Input: CSR-stored $R_{CSR}$ and CSR-stored $S_{sparse}$
Output: List of result tuples $(x, z)$
1 SPA[0..|$Z$|-1]=−1;
2 for $cur_x \leftarrow 0$ to $|X|$ do
3     foreach $cur_y$ related to $cur_x$ in $R_{CSR}$ do
4         foreach $cur_z$ related to $cur_y$ in $S_{sparse}$ do
5             if SPA[$cur_z$]!=$cur_x$ then
6                 SPA[$cur_z$]=$cur_x$;
7                 Result.append(($cur_x$, $cur_z$));
8 return Result;
```

all non-zero element S_{y_k,z_j} by visiting entries $Col[RowPtr[y_k]]$ .. $Col[RowPtr[y_k+1]-1]$. In fact, $RowPtr$ serves as the hash bucket header, and $Col[RowPtr[y_k]]$ .. $Col[RowPtr[y_k+1]-1]$ contain the hash entries in bucket y_k. In this way, we avoid the hash function computation and hash conflicts in common hash table designs.

Second, DHK performs deduplication using a z-vector. For each x_i, it initializes the z-vector to all zeros. Then, it checks all (x_i, y_k)s to compute the join results. For every result (x_i, z_j), DHK increments the corresponding element in the z-vector. Hence, the non-zero elements in the z-vector indicate the deduplicated Join-Project results. However, the initialization cost is $O(|Z|)$, while the number of non-zero z's can be small for the sparse part of the data. Consequently, the initialization of the z-vector is often a main cost of the deduplication computation. We remove this per-x initialization cost with a monotonically increasing flag for different x. As shown in Algorithm 2, the *SPA* array is the z-vector. We initialize the *SPA* array only once before any computation. The cur_y and cur_z loops (Line 3–7) compute the join results for the given cur_x. For a newly computed join result (cur_x, cur_z), we set $SPA[cur_z]$ to cur_x. If there are multiple duplicate (cur_x, cur_z), $SPA[cur_z]$ is set only once for the first instance. In this way, for the next $cur_x + 1$, the previous content of *SPA* is *automatically* invalid. This saves the cost of initializing *SPA* in every cur_x loop iteration.

We consider the time and space complexity of SparseBMM. Line 5 of Algorithm 2 runs $|OUT_J|$ times. Thus, the time complexity is $\Theta(|R| + |S| + |OUT_J|)$. While this is the same as the classical hash-based solution, SparseBMM reduces the constant factor, accelerating hash table visits and deduplication. Moreover, SparseBMM requires $\Theta(|R| + |S| + |Z|)$ space if the final output is consumed by upper level operators. While hash-based deduplication requires $\Theta(|OUT_P|)$ space for the hash table, SparseBMM allocates only $\Theta(|Z|)$ space for *SPA*, which is often much smaller than $\Theta(|OUT_P|)$.

2.5 DenseEC

To compute $C_{x_i,z_j} = \sum_{k=1}^{|Y|} R_{x_i,y_k} S_{y_k,z_j}$, standard dense MM enumerates all pairs of R_{x_i,y_k} and S_{y_k,z_j}. We observe that the computation can stop early as soon as there is a non-zero $R_{x_i,y_k} S_{y_k,z_j}$. Therefore, we propose a DenseEC algorithm to leverage this observation, as listed in Algorithm 3.

In Algorithm 3, the first two *for*-loops enumerate all the pairs of $R.x$ and $S.z$. Line 4–11 use one of two methods to check if there is any common y between $Bitmap_x$ and $Bitmap_z$. The first method uses SIMD to compute the bit-wise *AND* of $Bitmap_x$ and $Bitmap_z$ (e.g., using _mm256_testz_si256). The second method examines

Algorithm 3: DenseEC (for dense data).

```
Input: CSR-stored $R_{CSR}$ and bitmap array $S_{dense}$
Output: List of result tuples $(x, z)$
1 for $cur_x \leftarrow 0$ to $|X|$ do
2     $Bitmap_x \leftarrow$ Save the $y$ in the $cur_x$ row as bitmap;
3     foreach $cur_z$ in $S_{dense}$ do
4         if $f_3(m_x, m_z, |Y|)>0$ then /* SIMD is better */
5             if SIMD_AND($Bitmap_x$,$Bitmap_z$) not all 0 then
6                 Result.append(($cur_x$, $cur_z$));
7         else
8             foreach $y$ related to $cur_x$ in $R_{CSR}$ do
9                 if the yth bit in $Bitmap_z$ == 1 then
10                    Result.append(($cur_x$, $cur_z$));
11                    Break;
12 return Result;
```

each y related to cur_x in $Bitmap_z$ using a random memory access. If any common y is found, both methods stop early. Generally speaking, the SIMD method performs better when the cur_x row has a large number of y. We determine which method to use with function f_3, which will be discussed in Section 2.6.

Apart from the time saving due to early stopping, DenseEC saves memory space compared to dense MM. DenseEC represents every element as a single bit rather than a 4-byte integer or floating point value in BLAS packages. Moreover, a dense MM invocation would require space to save the temporary output matrix. In comparison, DenseEC never generates the output matrix.

2.6 Evaluation Path Section Functions

We compute the three functions used in the DIM^3 algorithm to select different evaluation paths. Table 1 lists the symbols used in this subsection. All the listed parameters can be measured in advance.

Strategy Selection (f_1). DHK chooses the classical solution using a rule-of-thumb condition: $|OUT_J| \leq 20 \cdot N$, where $N = |R| = |S|$. In comparison, DIM^3 determines whether to use the classical solution or the hybrid solution with function f_1. As the classical solution is beneficial only when the data is sparse, we compare mapping+SparseBMM and the classical solution to compute f_1. When $f_1 > 0$, the classical solution is faster and will be chosen.

$$\begin{aligned}
f_1(|R|,|S|,|OUT_J|) &= (T_{mapping} + T_{SparseBMM}) - (T_{join} + T_{dedup}) \\
&= T_{mapping} + (T_{SparseBMM} - T_{join}) - T_{dedup} \\
&\approx 2(|R| + |S|)t_{map} + |OUT_J|t_{randRW} - |OUT_J|t_{hash} \qquad (2)
\end{aligned}$$

Here, $T_{mapping} = 2(|R| + |S|)t_{map}$ from Section 2.2. To estimate $T_{SparseBMM} - T_{join}$, we see that generating join results with CSR in SparseBMM has smaller or similar cost compared to hash joins. Hence, the difference is mainly the deduplication cost of SparseBMM.

Table 1: Symbols used in Section 2.6.

Symbol	Description
t_{seqR}	time for sequential memory read
t_{randR}	time for random memory read
t_{randRW}	time for random memory read-modify-write
t_{hash}	time of a lookup or insertion to plain hash table
t_{map}	time to access the optimized hash table for mapping
t_{ECs}	time of a non-SIMD comparison in DenseEC
t_{ECd}	time of a SIMD comparison in DenseEC

This is $|OUT_J| * t_{randRW}$ because SparseBMM performs a random access to the SPA array per join result. Finally, $T_{dedup} = |OUT_J| t_{hash}$ because the classical solution performs a hash table access for deduplicating every join output tuple.

Bitmap Comparison in DenseEC (f_3). Line 4 of Algorithm 3 uses function f_3 to choose the SIMD or non-SIMD method for comparing $Bitmap_x$ and $Bitmap_z$. The bitmaps have $|Y|$ bits. Suppose there are m_x and m_z 1's in $Bitmap_x$ and $Bitmap_z$, respectively.

In the non-SIMD method, the comparison stops as soon as a check hits a set bit in $Bitmap_z$. The probability that the check hits a set bit is $p_s = \frac{m_z}{|Y|}$. The number of checks follows a geometric distribution with probability p_s, and the method performs at most m_x checks. Hence, the expected number of checks is calculated as:

$$Check_{nonsimd} = \sum_{i=1}^{m_x} i(1-p_s)^{i-1} p_s + m_x(1-p_s)^{m_x} = \frac{1-(1-p_s)^{m_x}}{p_s}$$

In the SIMD method, every SIMD comparison checks 256 bits in the two bitmaps. When the bits at the same position in the two bitmaps are both set, the comparison returns true and the process stops. The probability that an SIMD comparison returns true is $p_d = 1 - (1 - \frac{m_x m_z}{|Y|^2})^{256}$. The number of SIMD checks follows a geometric distribution with probability p_d, and the method performs up to $\frac{|Y|}{256}$ checks. Hence, the expected $Check_{simd} = \frac{1-(1-p_d)^{\frac{|Y|}{256}}}{p_d}$.

Then, we can compute f_3 as follows. When $f_3 > 0$, the SIMD method is preferred. Otherwise, the non-SIMD method is selected.

$$f_3(m_x, m_z, |Y|) = Check_{nonsimd} t_{ECs} - Check_{simd} t_{ECd} \qquad (3)$$

We can reduce the computation overhead of f_3 as follows. Note that m_x and $|Y|$ are constants in the for-loop at line 3 of Algorithm 3. $f_3(m_x, m_z, |Y|)$ is monotonically increasing with m_z. Therefore, an optimization is to use binary search to find the threshold value of m_{zt} so that $f_3(m_x, m_{zt}, |Y|) = 0$. Then, we can select the SIMD method if $m_z > m_{zt}$. Moreover, we find that multiple $R.x$'s can share the same m_x. Since m_{zt} is determined for a given m_x, we can cache the pairs of m_x and m_{zt}, then reuse the computed m_{zt} to avoid redundant binary searches. In this way, the time for computing all f_3 thresholds is at most 1.15ms for the real-world data sets in Section 5, which is less than 0.5% of the total run time.

Dense vs. Sparse Partitions (f_2). At line 6 in DIM^3 (Algorithm 1), intersection-free partitioning uses function f_2 to determine if $S.z$ column is dense or sparse.

For column z, the number of join results can be estimated as $\frac{m_z}{|S|}|OUT_J|$, where m_z is the number of S tuples in column z. We denote this value as $OUT_{J,z}$. We estimate the cost of processing column z using either SparseBMM and denseEC.

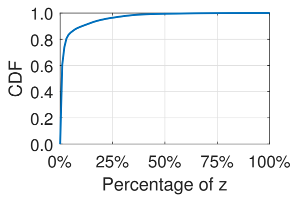
The cost of SparseBMM for z is computed as follows:

$$T_{sparseBMM} = \frac{(2|X|+|R|)t_{seqR} + 2|R|t_{randR}}{|Z|} + OUT_{J,z}(t_{seqR} + t_{randRW})$$

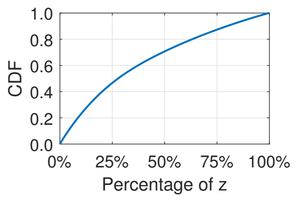
The first component computes the cost of the two for-loops at Line 2–3 amortized to one of $|Z|$ columns. The second component estimates the cost of Line 4–7. For DenseEC, we know that the cost for each pair of x and z from the above. Then we can sum this up to obtain $T_{denseEC} = \sum_{i=1}^{|X|} min(Check_{nonsimd} t_{ECs}, Check_{simd} t_{ECd})$.

Finally, we can compute f_2 as follows. When $f_2 > 0$, we consider column z as dense.

$$f_2 = T_{sparseBMM} - T_{denseEC} \qquad (4)$$



(a) HetRec data set.



(b) MV2 data set.

Figure 4: CDF of computation cost.

3 PARTIAL RESULT CACHING

We observe that computing Join-Project results for different z values can take widely different amounts of time for real-world data sets. Figure 4 depicts the CDF (Cumulative Distribution Function) of computation cost for two representative real-world data sets. In HetRec, the top 1% of z values take 60% of the total time. In MV2, the top 10% of z values contribute to 23% of the total cost. This motivates us to study caching the results of a subset of z values to improve Join-Project performance. Note that our technique is different from materialized views [29] or query result cache [28], where the full query results are cached.

DIM^3 with Cached Partial Results. We choose to cache results based on z values because the intersection-free partitioning method divides table S according to the $S.z$ values. Hence, it is easy to integrate the cached partial results into the DIM^3 algorithm.

Suppose Z_{cached} is the subset of $S.z$, whose Join-Project results are cached. That is, the cached partial results are $R_{cached} = \{(x,z) | (x,z) \in \Pi_{x,z}(R(x,y) \bowtie_y S(z,y)) \wedge z \in Z_{cached}\}$. In Figure 2, we can simply omit any $z \in Z_{cached}$ when generating S_{sparse} and S_{dense}, then keep the other steps of DIM^3 unchanged. Finally, we output R_{cached} in addition to the results computed by DIM^3.

Caching Score. Given a caching space budget B for a Join-Project query, we rank z values for caching with the following score:

$$CacheScore(z) = \frac{cost(z)}{space(z)}$$

$space(z)$ is the cache space required to store the (x,z) results for the given z. The cache content is $(z, size, x_1, x_2, ..., x_k)$. When k is small, $size$=k>0, and we cache the original results. When k>0.5$|X|$, we store the complement set of x's to save space, and set $size = k - |X|$<0. Hence, $space(z)$=2+$min(k, |X| - k)$.

$cost(z)$ is the time for computing (x,z) results for the given z. This is the benefit from caching z. Since the processing related to z is distributed into many inner loops in SparseBMM and DenseEC, we cannot directly measure $cost(z)$. Instead, we collect statistics and estimate the cost as follows:

$$cost(z) = n_{sparse}(t_{seqR} + t_{randRW}) + n_{simd} t_{ECd} + n_{nonsimd} t_{ECs}$$

where n_{sparse} is the number of times that z is checked in the inner loop of the SparseBMM algorithm, n_{simd} and $n_{nonsimd}$ are the number of times that z is encountered in the SIMD and non-SIMD part of the inner loop of the DenseEC algorithm, respectively.

$CacheScore(z)$ shows the benefit per unit space for caching z. The higher the $cost(z)$, the lower the $space(z)$, the more beneficial to cache z. Interestingly, a z value with high $cost(z)$ may produce a large number of results. This actually leads to small $space(z)$ when the complement result set is saved.

Practical Considerations. We discuss several practical issues for using the partial result caching. First, database users can enable Join-Project caching dynamically. We see in experiments that the statistics collection and computation for $cost(z)$ and $space(z)$ do not incur significant overhead for DIM^3. Hence, after caching is enabled, the first run of DIM^3 can compute $CacheScore(z)$ and populate the partial result cache. Then, subsequent runs of DIM^3 on the same tables can leverage the cached results to improve performance. Second, when the underlying tables are modified by insert/delete/update, we can simply invalidate the cache and let the next run of DIM^3 to re-populate the result cache. Note that updating the cached results (similar to the view maintenance problem [4]) is beyond the scope of this work. Finally, we can use the cached result to support filter predicates on x and/or z. However, if there are filter predicates on y, we have to run DIM^3 without the cached result.

4 SUPPORT FOR JOIN-OP QUERY TYPES

In the above, we focus on the Join-Project operation. Join-Project is a special case of Join-*op* queries, where *op* is a common SQL operation. An interesting question arises: Is it possible to apply DIM^3 to Join-*op* queries in general? We consider *op* other than projection in the following:

- *Group-by Aggregation*: The group-by operation inherently removes duplicates. A join followed by a group-by aggregation operation, which we call Join-Aggregate for simplicity, implicitly performs a Join-Project operation. We describe how to extend DIM^3 to support Join-Aggregate in Section 4.1.
- *Join*: Multiple join operations are followed by a duplicate eliminating projection. One strategy to evaluate this type of queries is to employ Join-Project to deduplicate the results of the last join operation. But can we do better? We consider how and when to push the deduplication down in the query plan of MJP in Section 4.2.
- *Selection or Sorting*: As Join-Selection or Join-Sorting do not require deduplication, there is no need to employ Join-Project for these types of queries in general. However, in special cases where there are too many duplicates, an alternative evaluation strategy can be more efficient. We compute Join-Project and keep the duplicate count for each generated join result tuple. Then, the selection or sorting operation processes the much smaller, deduplicated join result, thereby achieving better performance. Finally, we output the correct number of duplicates based on the per-tuple duplicate counts.
- *Intersection/Difference/Union*: First, the computation of intersection is similar to a join operation. Hence, for Join-Intersection, we can employ MJP. Second, set difference can be evaluated as a left-outer join followed by deduplication. We can modify DIM^3 to compute Outer-Join-Project for Join-Difference. To support outer-joins, DIM^3 can be extended with a bitmap for x (z). It sets a bit if the corresponding x row (z column) has generated join results. In this way, the modified DIM^3 can compute x (z) with no matches for outer-joins. Finally, Join-Union requires the deduplication of the outputs from two joins. We can push the deduplication operation down in the query plan, and apply similar considerations as in Section 4.2.

4.1 Join-Aggregate

Our DIM^3 algorithm can be applied to Join-Aggregate operations with slight modifications. Without loss of generality, we divide Join-Aggregate operations into two categories based on the group by attributes: i) group-by attributes are from both tables, and ii) group-by attributes come from one table.

Group-by Attributes from Both Tables. For instance, given tables $R(x, y, v)$ and $S(z, y, u)$, we want to compute

$$_{x,z}\mathcal{G}_{aggregate(f(R.v,S.v))}(R(x, y, v) \ltimes_y S(z, y, v))$$

This task is similar to the original Join-Project operation. The main difference is that it computes $aggregate(f(R.v, S.u))$ on the join results with the same (x, z) rather than deduplicating the results.

We modify DIM^3 to support this task. In SparseBMM, we change the SPA array to contain the aggregate for each cur_z. SPA is initialized in each outer-loop iteration. Line 5–7 is modified to accumulate the aggregate for group (cur_x, cur_z). In DenseEC, bitmaps cannot support the aggregates. Therefore, we use a plain dense MM, while computing aggregates for each pair of (cur_x, cur_z).

Group-by Attributes from One Table. Given tables $R(x, y)$ and $S(z, y)$, we want to compute the following query. If the group-by attribute is from S, we switch table R and S.

$$_x\mathcal{G}_{aggr(z)}(R(x, y) \ltimes_y S(z, y))$$

We can rewrite the query as follows:

$$_x\mathcal{G}_{aggr''(z')}(R(x, y) \ltimes_y (_y\mathcal{G}_{aggr'(z)}(S(z, y))))$$

We first compute the group-by aggregate on S with y as the group-by key. If $aggr$ is sum, min, or max, then $aggr'$ and $aggr''$ are the same. For $count$, $aggr'$ is $count$ and $aggr''$ is sum. For avg, the $aggr'$ consists of both sum and $count$. Then $aggr''$ accumulates the two components, and finally computes a division to obtain the avg. We denote the resulting table as $SG(y, z')$, where z' is the intermediate aggregate value(s) for y. Note that $|SG| = |Z|$ is often much smaller than $|S|$. SG is highly sparse because there is no duplicate y's in the table. Therefore, we employ the classical hash-based algorithm to join R and SG then compute the final group-by aggregates.

4.2 MJP (Multi-Way Joins with Projection)

So far, we have been focusing on Join-Project on *two* tables. In this subsection, we study deduplication on the results of joining *multiple* tables. This is also called conjunctive queries with projection [11]. For example, the line join projection with n tables is expressed as:

$$\Pi_{x_1,x_{n+1}}(R_1(x_1, x_2) \bowtie_{x_2} R_2(x_2, x_3) \bowtie_{x_3} ... \bowtie_{x_n} R_n(x_n, x_{n+1}))$$

Figure 5 illustrates the query plan tree for evaluating a MJP query. The deduplication operation (Π) can be pushed down to after each join operation. While it incurs extra overhead, deduplication reduces the intermediate result size, thereby reducing the cost of subsequent join operations. There are two baseline execution plans. The first plan computes all the joins and then deduplicate the final join results. The second plan deduplicates the results immediately after each join. However, it is easy to construct cases to show neither plan is optimal. Therefore, we need to judiciously place the deduplication operations into the query plan tree.

In the following, we develop a DP (Dynamic Programming) algorithm to find the optimal query plan. To limit the scope of our investigation, we make the following assumptions.

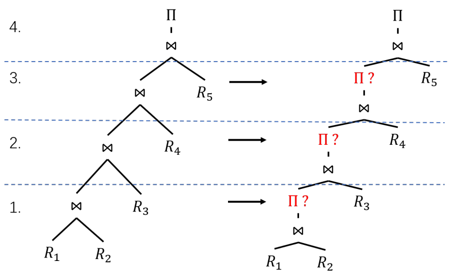


Figure 5: The deduplication placement problem.

(1) *The join order is given by the query optimizer.* We focus on the deduplication placement problem. For ease of presentation, we number the tables in the join order as R_1, ..., R_k. The problem of optimizing both join order and deduplication placement is beyond the scope of this work.

(2) *The query plan is in the form of a left-deep tree [21, 39].* Left-deep or right-deep trees are widely used in RDBMSs to process multi-way joins. If a solution produces left-deep trees, it is easy to modify it to support right-deep trees.

(3) *Estimated $|OUT_{Ji}|$ and $|OUT_{Pi}|$, where i=2,...,k, are available.* (Please refer to recent work on cardinality esitmation for details [6, 16, 31, 37].) Here, $|OUT_{Ji}| = |R_1 \bowtie \cdots \bowtie R_i|$ denotes the size of the intermediate results after joining the first i tables without deduplication. $|OUT_{Pi}| = |\Pi(OUT_{Ji})|$ denotes the size of the intermediate results after deduplication. For simplicity, we define $|OUT_{P1}| = |OUT_{J1}| = |R_1|$.

(4) *Adding deduplication to the i-th join reduces the final join result by a factor of $\frac{|OUT_{Pi}|}{|OUT_{Ji}|}$.* That is, $|\Pi(R_1 \bowtie \cdots \bowtie R_i) \bowtie \cdots \bowtie R_k| \approx \frac{|OUT_{Pi}|}{|OUT_{Ji}|}|OUT_{Jk}|$. This is reasonable since the join input size to the $(i+1)$-th join is reduced by a factor of $\frac{|OUT_{Pi}|}{|OUT_{Ji}|}$. This assumption simplifies the estimation of the final result size after inserting a deduplication operation. More precise estimation requires collecting more statistics, and may incur much higher cost.

We use DP_i to denote the optimal time for computing $\Pi(R_1 \bowtie \cdots \bowtie R_i)$. We observe that if we already add deduplication to the i-th join, the deduplicated result OUT_{Pi} does not change if more deduplication operations are added to the joins before the i-th join. Therefore, we have the following equations.

$$DP_i = \begin{cases} 0, & i = 1 \\ \min_{j=1}^{i-1}(DP_j + \sum_{h=j+1}^{i-1} JoinCost_h + JPCost_i), & i > 1 \end{cases}$$

The formula inside *min* computes the case where a deduplication is added to the j-th join and there is no deduplication from the $(j+1)$-th to $(i-1)$-th join. The total cost of this case has three components: (i) DP_j, which is the optimal time for computing OUT_{Pj}; (ii) the cost of the subsequent joins, i.e., the $(j+1)$-th to $(i-1)$-th join; and (iii) the cost of the final join project operation. Note that when j=1, there is no deduplication before the i-th join.

Based on assumption (3), we estimate the input table size for component (ii) and (iii). Specifically, $JoinCost_h$ is the cost of joining $\Pi(R_1 \bowtie \cdots \bowtie R_j) \bowtie \cdots \bowtie R_{h-1}$ and R_h. The size of the former is estimated as $\frac{|OUT_{Pj}|}{|OUT_{Jj}|}|OUT_{Jh-1}|$. Moreover, $JPCost_i$ is the cost of the Join-Project operation at the i-th join. The sizes of the two input tables to the Join-Project operation are $\frac{|OUT_{Pj}|}{|OUT_{Jj}|}|OUT_{Ji-1}|$ and $|R_i|$. The intermediate join result in the Join-Project operation can be estimated as $\frac{|OUT_{Pj}|}{|OUT_{Jj}|}|OUT_{Ji}|$ and the final output size is $|OUT_{Pi}|$. Given the sizes of the input tables, the intermediate join result, and the final output, we can use the formulas in Section 2.6 to estimate the cost of the Join-Project operation.

For a MJP query on n tables, the path to get DP_n in the DP process gives the optimal deduplication placement. The time complexity of this algorithm is $\Theta(n^3)$. As n is often not large, the overhead of this DP algorithm is small.

5 PERFORMANCE EVALUATION

In this section, we evaluate the performance of our proposed solutions using both real-world and synthetic data sets.

5.1 Experimental Setup

Machine Configuration. All experiments are performed on a machine with Intel Core i7-9700 CPU (3.00GHz, Turbo Boost 4.70 GHz, 8 cores/8 threads, 12MB last level cache) and 32 GB RAM, running Ubuntu 18.04.5 LTS with Linux 5.4.0-81 kernel. All code is written in C/C++ and compiled with *g++* 7.5.0 using *–std=c++11*, *-O3*, and *-mavx* flags. MKL (Intel Math Kernel Library) is used in DHK. By default, we run single-threaded experiments. In the parallel experiments, we use OpenMP for parallelization.

Datasets. We use six real-world datasets with different input size, $|OUT_J|$, and $|OUT_P|$ in our experiments, as shown in Table 2. Amazon [32] data set records the frequently co-purchased products on Amazon website. The Join-Project queries help to find potential products that can be co-purchased. Slashdot [33] is a technology-related news website. The data set contains friend/foe links between users of Slashdot. The query computes the indirectly connected pairs of users. The data set and the query of HetRec [7] follow the motivating example in Section 1. MV1 and MV2 are two MovieLens [15] data sets. They contain user ratings for movies. Similarly, Jokes [13] contains user ratings for jokes. The Join-Project queries on MovieLens and Jokes data sets find users that have rated the same objects. Friendster [45] is a social network dataset that contains the retweet relationship between users. We use this data set to study MJP, which computes the multi-hop connections between users.

We also conduct experiments on the TPC-H data set with SF=10. A representative query joins *LineItem* and *Orders*, then projects on *CUSTKEY* and *SUPPKEY* to attain the purchase relationship between customers and suppliers. However, since this join is a primary-foreign key join, $|OUT_J|$ is similar to the size of *LineItem*. Both DIM^3 and DHK choose the classical solution, showing the same performance. Hence, we do not study TPC-H further.

In contrast, $|OUT_J|$ and $|OUT_P|$ in the other data sets are much larger than their input sizes. The higher the *pnz*, the denser the dataset, and thus the more intermediate join results. As we do not assume any special sort order in the input tables of Join-Project, we randomly shuffle the data sets before the experiments.

Solutions to Compare. We compare two categories of solutions: stand-alone Join-Project algorithms, and full-fledged RDBMSs.

We compare the following stand-alone implementations. (1) *Classical*: The classical solution performs the Radix join [2] then a

Table 2: Real-world data sets used in experiments.

Data set	$\|R\|$	$\|S\|$	$pnz(R)$	$pnz(S)$	$\|OUT_J\|$	$\|OUT_P\|$
Amazon	1.2M	1.2M	0.0018%	0.0018%	14M	11M
Slashdot	905K	905K	0.02%	0.02%	118M	81M
HetRec	487K	438K	0.02%	0.03%	125M	34M
MV1	500K	500K	2.2%	2.2%	204M	29M
MV2	1M	1M	4.5%	4.5%	816M	35M
Jokes	617K	617K	25%	25%	10B	622M
Friendster	1.8B		4.2×10^{-7}		–	–

Note: $pnz(M)$ is the percentage of non-zero elements in matrix M.

hash-based deduplication using a flat_hash_map [40]. (2) *MKL*: The state-of-the-art dense MM in Intel MKL is used to evaluate Join-Project. We implement the baseline mapping algorithm as described in Section 2.2. (3) *DHK*: We obtained the DHK code from the authors of the DHK paper [10]. Algorithm 3 in DHK chooses the classical solution when $|OUT_J| \leq 20 \cdot N$, where $N = |R| = |S|$. We find this feature is missing in the DHK code, and implement the feature. The code assumes the input contains consecutive natural numbers. Hence, we add the baseline mapping algorithm. The code leaves the final deduplication unimplemented. Thus, we add a deduplication step that checks every result (x,z) from the sparse part against the matrix $C^{x \times z}$ computed in the dense part. (4) *DIM*^{3}: Our proposed solution follows the description in Section 2. We also study the individual components of DIM^3. For the MM component, we compare DenseEC, SparseBMM, MM in MKL, and a sub-cubic MM [3] based on the Strassen algorithm [41].

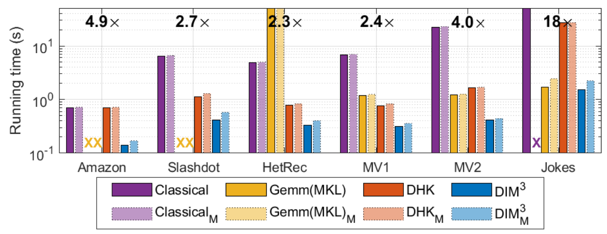
In addition to the stand-alone algorithms, we compare DIM^3 with four full-fledged RDBMSs. (5) *DBMSX*: one of the best performing commercial RDBMSs. (6) *PostgreSQL* version 13.3 and (7) *MariaDB* version 10.5.11: two popular open-source RDBMSs. (8) *MonetDB* version 11.39.17: a representative analytical main memory RDBMS. We set the configuration parameters of the RDBMSs to ensure that they make full use of the memory. For each experiment, we run the same query on the target data set twice. The first run warms up the RDBMSs so that the input tables are loaded into main memory. Then we measure the performance of the second run.

Our comparison between *DIM*^{3} and RDBMSs is meaningful. One concern is that RDBMSs pay additional cost, including socket communication, SQL parsing, query optimization. (Note that we compute a final count to avoid returning a large number of query results from RDBMSs.) We quantify this additional cost by measuring the execution time of the same queries on a very small data set. The result is 1–3ms. In comparison, our reported run times on RDBMSs are from 7s to about 1 hour. Hence, the extra cost of 1–3ms is negligible. Our reported run times indeed correspond to the cost of Join-Project query processing in RDBMSs.

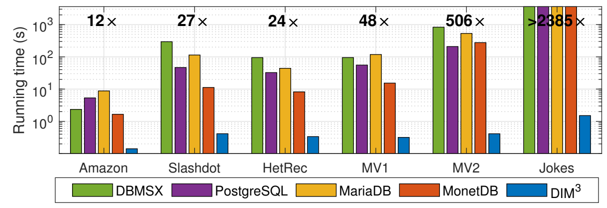
Unless otherwise noted, we focus on single-threaded performance in the experiments. The scalability results report multi-threaded performance to show that our proposed algorithms are amenable to parallelization. Partial result caching is disable by default. Every experiment is run five times. We report the average of the five runs.

5.2 Evaluation for Join-Project Operations

Performance of Stand-alone Solutions on Real-World Data Sets. Figure 6(a) compares DIM^3 with stand-alone Join-Project algorithms on real-world data sets. We see that DIM^3 achieves the best performance among all stand-alone algorithms with or without result



(a) Comparison with stand-alone Join-Project algorithms.



(b) Comparison with RDBMSs.

Figure 6: Join-Project on real-world datasets. (By default, the algorithms count then discard the result tuples. Subscript 'M' denotes a version of an algorithm that materializes the result tuples in memory. 'X' shows when MKL/Classical runs out of memory and fails. We label speedups of DIM^3 over DHK in (a), and over the best RDBMS solution in (b).)

materialization. Compared to *DHK*, the state-of-the-art algorithm, DIM^3 achieves 2.3×-18× improvements. For all the six data sets, DIM^3 chooses the hybrid strategy. In contrast, for Amazon, DHK chooses the classical solution. The speedup of DIM^3 over DHK for Amazon shows that our strategy selection function f_1 is more accurate than the rule-of-thumb condition of DHK.

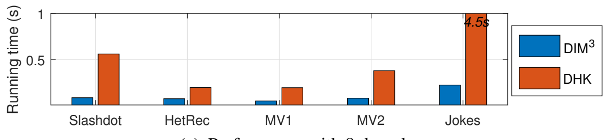
Moreover, DIM^3 has smallest memory footprints among all algorithms, as shown in Table 3. We use '*/usr/bin/time -v*' to measure the peak memory usage (i.e., maximum resident set size). Typically, the final Join-Project results will be consumed by upper-level operators, and therefore we do not allocate space for storing the final results. Compared to *Classical*, when the data sets are dense, DIM^3 saves 80%–99% memory because it chooses the hybrid strategy and saves the space required by hash-based deduplication in *Classical*. The memory usage of *Gemm(MKL)* is mainly determined by the matrix sizes. MKL runs out of memory for the two sparsest data sets (i.e., Amazon and Slashdot). When the datasets are dense, DIM^3 saves 68%–99% memory of *Gemm(MKL)* because DenseEC saves the space of the output matrix, and uses one bit per element rather than 4-byte integers in MKL. As for *DHK*, one main source of its memory usage is the input and output matrices for dense MM invocation. In comparison, DenseEC significantly saves this memory.

Comparison with RDBMSs on Real-World Data Sets. Figure 6(b) compares DIM^3 with RDBMSs. We see that DIM^3 outperforms all the RDBMSs. For data sets with $|OUT_J| \gg |R| + |S|$, DIM^3 achieves one to three orders of magnitudes of speedups.

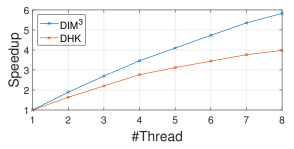
We examine the query plans generated by the RDBMSs and see that they all essentially employ the *Classical* solution, i.e., a join followed by a deduplication of the intermediate results. Among the RDBMSs, MonetDB is the best when memory is sufficient. However, its performance drops sharply when the required space exceeds the memory size (for MV2 and Jokes). In the case of Jokes, all the RDBMSs fail to perform the deduplication operation entirely in

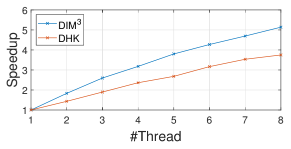
Table 3: Memory usage of stand-alone Join-Project algorithms.

	Amazon	Slashdot	HetRec	MV1	MV2	Jokes
DIM^3	86MB	64MB	32MB	39MB	72MB	47MB
DHK	426MB	70MB	168MB	68MB	132MB	2.4GB
Gemm(MKL)	>32GB	>32GB	7.1GB	211MB	227MB	2.4GB
Classical	426MB	3.0GB	1.5GB	789MB	1.5GB	24GB



(a) Performance with 8 threads.





(b) Scalability on the HetRec data set. (c) Scalability on the MV1 data set.

Figure 7: Multi-threaded performance.

memory. The queries take much longer time because of the disk I/Os for storing and accessing the intermediate results. We stop the queries when they took longer than 1 hour. The figure reports the lower bound (i.e., 3600s) for RDBMSs on the Jokes data set.

As RDBMSs implement the classical solution, we focus on the comparison with stand-alone solutions in the rest of this subsection.

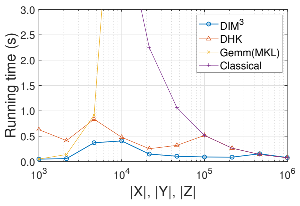
Scalability. We implement the multi-threaded DIM^3 with OpenMP by modifying around 20 lines of code. Specifically, we parallelize the outer *for*-loop in SparseBMM and DenseEC. Since every outer loop iteration focuses on a specific x, there is no dependence or contention among the execution of different outer-loop iterations. While more involved changes may better parallelize the algorithm, we find that such simple modifications can already achieve promising results.

We compare DIM^3 with *DHK* using 8 threads in Figure 7(a). We see that DIM^3 achieves 2.6×-20× speedups over DHK. Figure 7(b)–(c) show the scalability of the two algorithms. The Y-axis reports the speedup compared to the single-threaded execution of the same algorithm. We see that DIM^3 shows good scalability as the number of threads increases from 1 to 8.

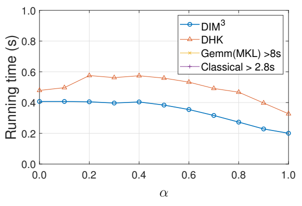
Sensitivity Analysis with Synthetic Data Sets. We use synthetic data sets to investigate the Join-Project performance for a wide range of situations. The default parameters in these experiments are as follows. All input columns are generated with uniform distributions. $|R| = |S| = 10^6$, and $|X| = |Y| = |Z| = n = 10^4$. We study the performance impact varying n, data skews, and selectivity in Figure 8. Overall, we see that DIM^3 performs the best in all the cases.

(1) Varying n. Figure 8(a) varies n, the number of distinct $x/y/z$, while keeping the number of non-zero elements (i.e. $|R|$ and $|S|$) fixed. As n increases, the input data sets become more and more sparse, and $|OUT_J|$ decreases from 10^9 to 10^6. *Classical* runs faster for more sparse data sets. *Gemm(MKL)* sees a sharp increase of run time as the matrix dimensions (i.e., n) increase. DIM^3 wisely selects evaluation strategies based on data density. It employs DenseEC for all data when $n<10^4$, SparseBMM when $n \in [10^4, 4 \times 10^5]$, and the classical solution when $n>4 \times 10^5$. In this way, DIM^3 achieves the best performance in all cases.

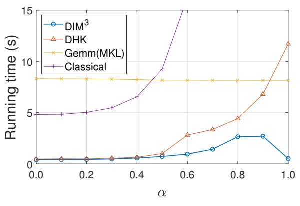
DHK has good performance in the middle range in Figure 8(a) and switches to Classical in the higher range. However, its performance



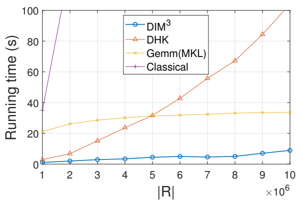
(a) Varying n ($|X| = |Y| = |Z| = n$).



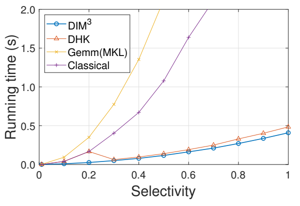
(b) Zipf on $R.x$ and $S.z$.



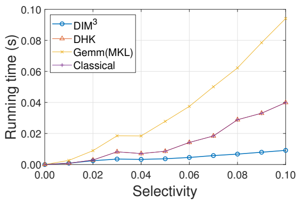
(c) Zipf on $R.y$ and $S.y$.



(d) R-mat.



(e) Varying selectivity of $R.x$ and $S.z$.



(f) Zoom in to bottom left of (e).

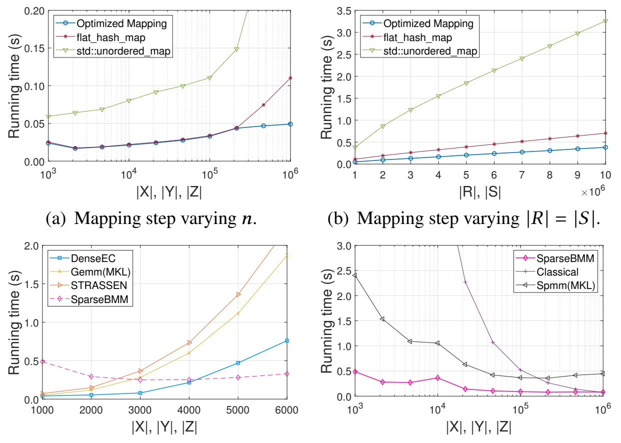
Figure 8: Sensitivity analysis of DIM^3.

is worse than *Gemm(MKL)* in the lower range. To understand the reason, we look into the *DHK* code and find that *DHK* restricts the degree thresholds (i.e., Δ_1 and Δ_2 [10]) to have $\Delta_2 = \frac{|R|\Delta_1}{|OUT_P|}$ in order to reduce the threshold search space. Unfortunately, this narrows the range of considered thresholds. For the lower range when the data is very dense, *DHK* fails to put all data into the dense part, and thus performs worse than *Gemm(MKL)*. In comparison, DIM^3 does not suffer from this problems. DIM^3 makes the partition decision based on z, which is simple and less error-prone to compute.

(2) Varying data skews. In Figure 8(b), we generate $R.x$ and $S.z$ using the Zipf distribution, and vary the parameter α from 0 to 1. As $R.x$ and $S.z$ become more skewed, $|OUT_J|$ stays around 10^8, while the deduplicated result size $|OUT_P|$ decreases from 6×10^7 to 1.4×10^7. DIM^3 partitions the more skewed z's into the dense part as α increases, and achieves a speedup of ~1.5× compared to *DHK*.

In Figure 8(c), we generate the join attribute, $R.y$ and $S.y$, using Zipf distribution, and vary α from 0 to 1. As $R.y$ and $S.y$ become more skewed, the number of intermediate join results ($|OUT_J|$) increases sharply from 10^8 to 4.3×10^9, and the data sets become increasingly dense. *Classical* runs slower as the data sets become denser. *Gemm(MKL)* sees a flat curve because the matrix dimensions are the same. *DHK* suffers from the threshold search space problem for very dense data sets. DIM^3 achieves the best performance in all cases. There is a dip in the DIM^3 curve at α=1 because early stopping effectively reduces DenseEC computation in this case.

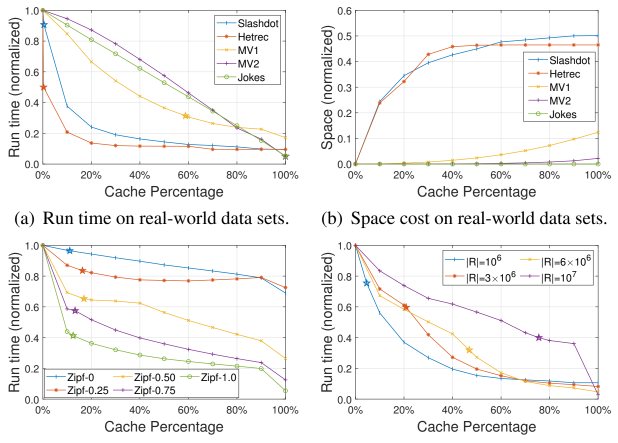
In Figure 8(d), we use R-mat [8] to generate skew data so that the two columns in a table are correlated. We use the R-mat parameters in Graph500 [35] (i.e., $a = 0.57$, $b = 0.19$, $c = 0.19$, and $d = 0.05$). We generate a directed graph with n vertices and $|R|$ edges. The Join-Project finds 2-hop paths in the graph. As R-mat requires n to be a power of 2, we set $n = 2^{14}$, which is close to the default 10^4. We vary



(a) Mapping step varying n. (b) Mapping step varying $|R| = |S|$.

(c) DenseEC varying n. (d) SparseBMM varying n.

Figure 9: Effectiveness of DIM^3 Components.



(a) Run time on real-world data sets. (b) Space cost on real-world data sets.

(c) Run time on Zipf data. (d) Run time on R-mat data.

Figure 10: Partial result caching. (★ shows the point where cached result size = input data size.)

$|R|$ from 10^6 to 10^7. From Figure 8(d), we see that DIM^3 performs significantly better than the other algorithms. As $|R|$ increases, the data set becomes increasingly dense. *Classical*, *Gemm(MKL)*, and *DHK* curves show similar trends as in Figure 8(c).

(3) Varying selectivity. In Figure 8(e), we consider the cases where there are filtering predicates on both $R.x$ and $S.z$. We vary their selectivity at the same time from 0 to 1. Figure 8(f) zooms in to the bottom left of the Figure 8(e). As the selectivity decreases, both the input sizes and the output size decrease. Hence, all the algorithms run faster. From the figure, we see that DIM^3 performs the best in all cases. Note that when the selectivity is very small (less than 0.03), DIM^3 switches to the classical solution. When the selectivity is less or equal than 0.2, *DHK* switches to the classical solution.

Effectiveness of DIM^3 Components. In the following, we use synthetic data sets with the same default parameters as the above.

(1) Mapping step. We compare our optimized mapping solution with the baseline using std::unordered_map, and an improved baseline using flat_hash_map. In Figure 9(a), we fix $|R|$=$|S|$=10^6 while varying n from 10^3 to 10^6. The hash table size increases as n. When $n < 4 \times 10^5$, the hash table fits into the L3 cache, and thus we do not perform cache partitioning. Our solution has similar performance compared to flat_hash_map. When $n \geq 4 \times 10^5$, we use cache partitioning to reduce expensive CPU cache misses, thereby significantly out-performing flat_hash_map. As for std::unordered_map, it is much slower because of chained hashing.

In Figure 9(b), we fix n=10^6 while varying $|R|$=$|S|$ from 10^6 to 10^7. The size of the hash table is fixed. The number of hash visits increases. Hence, we see the increasing trends for all curves. Overall, our optimized mapping solution performs the best.

Moreover, we compare DIM^3 (with optimized mapping) and DIM^3 with baseline std::unordered_map for the Amazon data set. The mapping step takes 54.9% of the Join-Project run time for DIM^3 with baseline mapping. Replacing the baseline mapping with optimized mapping, DIM^3 achieves an improvement of 1.5x.

(2) DenseEC. In Figure 9(c), we vary n from 1000 to 6000, where the data sets are relatively dense and thus dense MM makes sense. We compare DenseEC with dense MM implementations.

We investigate sub-cubic fast MM algorithms. While the best known $O(n^{2.373})$ algorithm [12] is considered impractical because of its huge constant factors [30, 36], recent work aims to make the Strassen algorithm practical [3]. We obtain and evaluate this *STRASSEN* implementation [3]. Our initial results show that it is slower than Intel MKL. One potential reason is that the original *STRASSEN* supports only floating point MM, while the MKL run uses integer MM. Since the *STRASSEN* code calls BLAS as the underlying MM sub-routine, it is straight-forward to modify the code to call MKL's integer MM. The resulting integer *STRASSEN* indeed runs slightly faster, but it is still slower than MKL. Figure 9(c) shows the performance of the improved integer *STRASSEN*.

Overall, DenseEC out-performs dense MM implementations because early stopping can effectively reduce computation cost.

(3) SparseBMM. In Figure 9(d), we vary n from 10^3 to 10^6. The generated data sets become more and more sparse, and therefore sparse MM and *Classical* are more suitable. We compare SparseBMM with *Classical* and MKL's sparse MM. From the figure, we see that SparseBMM performs the best.

We also plot the SparseBMM curve in Figure 9(c). The crossing point of SparseBMM and DenseEC is around 4000. This shows why DIM^3 chooses between DenseEC and SparseBMM with function f_2.

5.3 Evaluation for Partial Result Caching

We evaluate partial result caching for DIM^3 using both real-world and synthetic data sets, as shown in Figure 10. The synthetic data sets use the same default parameters as in Section 5.2. Our solution retrieves cached partial results and performs JoinProject between R and $\{(z, y)|(z, y) \in S \wedge z \notin Z_{cached}\}$. The X-axis varies the percentage of z values that are cached. The Y-axis reports either the run time normalized to DIM^3 without caching, or the cached result size normalized to the size of the total JoinProject results.

Figure 10(a) and 10(b) show the run time and the cached result size on real-world data sets, respectively. For Slashdot and HetRec, the computation time distributions for z values are very skewed. The space cost is relatively high because most results are cached by their original values. Caching a relatively small percentage (e.g., 20%) of z values can significantly speedup DIM^3. On the other hand, the denser data sets (i.e., MV1, MV2, and Jokes) see less skewed computation time distributions for z values. As the number of results for the same z is often large, our technique to cache the complement set of result vectors can drastically reduce the cached result size. For example, the space at 100% (i.e., for caching all results) is only 12%, 6%, and less than 1% of the total result sizes for MV1, MV2, and Jokes, respectively. Thus, we can use a small amount of cache space to achieve significant speedups. Overall, if we cache either 20% of z values or up to the input data size, partial result caching achieves 3.3x–20x improvements over DIM^3 without caching.

Figure 10(c) shows the run time on synthetic data sets where $R.x$ and $S.z$ follow Zipf distribution. When Zipf parameter $\alpha = 0$, the data is uniformly distributed, and the benefits of caching is low. As α increases from 0.25 to 1, the computation time distribution for z values becomes more skewed and caching is more helpful. Caching only 10% of z values reduces the run time by 13% to 57%.

Figure 10(d) shows the run time on R-mat data varying the input table size $|R|$ from 10^6 to 10^7. If we limit the cache space to the size of the input table, partial result caching reduces the run time of DIM^3 by 25%–67%.

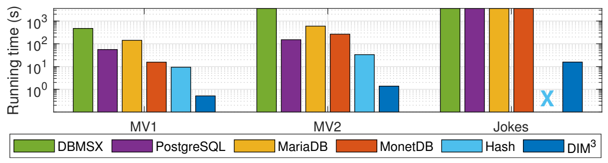
5.4 Evaluation for Join-OP Query Types

Join-Aggregate. Figure 11(a) compares the Join-Aggregate performance of DIM^3 with RDBMSs and the stand-alone hash-based solution on MV1, MV2, and Jokes. We focus on the more complex case where the group-by attributes are from both tables. As described in Section 4.1, if the group-by attributes are from one table, we can rewrite the query and employ the classical solution for efficient evaluation. We compute aggregates on the ratings for the MovieLens and Jokes data sets. (We omit Slashdot and HetRec in this set of experiments because they do not have natural value columns to express the aggregation.) We see that DIM^3 performs the best among all solutions. Compared with the hash-based solution, DIM^3 obtains a speedup of 18–24×. Compared with the fastest RDBMS solution, DIM^3 obtains a speedup of as least 30×.

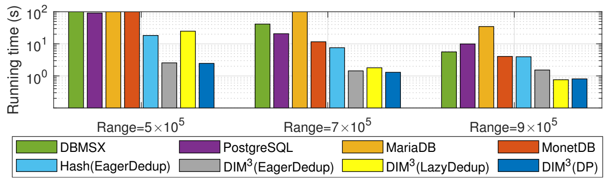
MJP. We evaluate the effectiveness of our proposed DP algorithm. We compare three DIM^3 variants with RDBMSs: 1) DIM^3*(DP)* runs DP to find the optimal query plan; 2) DIM^3*(EagerDedup)* places a deduplication after every join; and 3) DIM^3*(LazyDedup)* performs only one deduplication after all the joins.

In this set of experiments, we use the Friendster data set [45] and compute multi-hop connections between users. Since the original Friendster data set is large and very sparse, as shown in Table 2, we construct subsets of the data set using a parameter *range*. Given a *range*, we filter out any tuples whose attributes are both larger than *range*. From the remaining tuples, we randomly extracted 10 tables, each with 10^6 tuples, then perform a 10-way join with projection. We set *range*= 5×10^5, 7×10^5 or 9×10^5. Note that the larger the *range*, the sparser the input tables. For the three sub data sets, the final output size $|OUT_P|$ is 46M, 18M, and 8M, and DIM^3*(DP)* decides to place 4, 3, and 2 deduplication operations, respectively.

As shown in Figure 11(b), all DIM^3 variants run faster than RDBMSs because of the better Join-Project performance of DIM^3. Compared with the hash-based solution with EagerDedup strategy, DIM^3(DP) obtains a speedup of 4.9–7.4×. When *range*= 5×10^5, DIM^3*(DP)* gains a 10x speedup compared to *LazyDedup*. When *range*= 7×10^5, *DP* are 11% and 39% better than *EagerDedup* and *LazyDedup*, respectively. When *range*= 9×10^5, *DP* is 1.9x as fast as *EagerDedup*. However, *DP* is slightly (i.e., 5%) slower than



(a) Join-Aggregate. $(_{x,z}\mathcal{G}_{avg(|R.v-S.v|)}\,(R(x, y, v) \ltimes_y S(z, y, v)))$



(b) MJP.

Figure 11: Evaluation for Join-*op* query types.(*Hash*: stand-alone implementation using flat_hash_map for join, aggregation, and deduplication. 'X': *Hash* runs out of memory and fails.)

LazyDedup. In this case, *DP* places a deduplication after the first join. The hope is that the amount of computation in subsequent joins will be proportionally reduced, but the actual reduction is not as significant. Overall, we see that DIM^3*(DP)* performs the best among the three variants. This confirms the effectiveness of the DP algorithm for finding the optimal plan for evaluating MJP.

6 CONCLUSION AND FUTURE WORK

In this paper, we propose DIM^3 that combines intersection-free partitioning, optimized mapping, and DenseEC and SparseBMM algorithms to improve the state-of-the-art DHK solution. Moreover, we investigate partial result caching and extend DIM^3 to efficiently compute Join-*op* queries. Our results show that DIM^3 is a promising solution for the widely used Join-Project operation.

Several promising future directions remain to be explored. The first is out-of-core computation. When the input table R and S are too large to fit in the allocated memory, one way is to perform I/O partitioning for R and S according to $R.x$ and $S.z$, respectively. Then, we load each pair of $R.x$ partition and $S.z$ partition into memory and employ DIM^3 to compute the results. An alternative way is to partition the tables according to the join key y. However, a final deduplication step is necessary because the intermediate results from different partitions may contain duplicates.

The second direction is to study caching for multiple queries when partial result caching is enabled. In addition to traditional considerations, such as query access statistics, the cache space allocated to a Join-Project query becomes a tunable parameter. This adds a new dimension in the design of the caching strategy.

Last but not least, it is interesting to exploit new hardware to accelerate database operations. For example, GPUs (Graphics Processing Units) [5] and TPUs (Tensor Core Units) [19] can significantly accelerate matrix multiplication in Join-Project. NVM (Non-Volatile Memory) provides an interesting combination of persistence, capacity, and performance, which can be used to speed up queries with huge memory consumption [42].

ACKNOWLEDGMENTS

This work is partially supported by Natural Science Foundation of China (62172390). Shimin Chen is the corresponding author.

REFERENCES

[1] Rasmus Resen Amossen and Rasmus Pagh. 2009. Faster join-projects and sparse matrix multiplications. In *Proceedings of the 12th International Conference on Database Theory*. 121–126.

[2] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M Tamer Özsu. 2013. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endowment* 7, 1 (2013), 85–96.

[3] Austin R Benson and Grey Ballard. 2015. A framework for practical parallel fast matrix multiplication. *ACM SIGPLAN Notices* 50, 8 (2015), 42–53.

[4] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. 1986. Efficiently Updating Materialized Views. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 28-30, 1986*. ACM Press, 61–71.

[5] Sebastian Breß, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. 2014. Gpu-accelerated database systems: Survey and open challenges. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XV*. Springer, 1–35.

[6] Walter Cai, Magdalena Balazinska, and Dan Suciu. 2019. Pessimistic cardinality estimation: Tighter upper bounds for intermediate join cardinalities. In *Proceedings of the 2019 International Conference on Management of Data*. 18–35.

[7] Iván Cantador, Peter Brusilovsky, and Tsvi Kuflik. 2011. Second workshop on information heterogeneity and fusion in recommender systems (HetRec2011). In *Proceedings of the fifth ACM conference on Recommender systems*. 387–388.

[8] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 442–446.

[9] Steven Dalton, Luke Olson, and Nathan Bell. 2015. Optimizing sparse matrix—matrix multiplication for the gpu. *ACM Transactions on Mathematical Software (TOMS)* 41, 4 (2015), 1–20.

[10] Shaleen Deep, Xiao Hu, and Paraschos Koutris. 2020. Fast join project query evaluation using matrix multiplication. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1213–1223.

[11] Shaleen Deep, Xiao Hu, and Paraschos Koutris. 2021. Enumeration Algorithms for Conjunctive Queries with Projection. In *24th International Conference on Database Theory, ICDT 2021, March 23-26, 2021, Nicosia, Cyprus (LIPIcs)*, Ke Yi and Zhewei Wei (Eds.), Vol. 186. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 14:1–14:17.

[12] François Le Gall and Florent Urrutia. 2018. Improved rectangular matrix multiplication using powers of the Coppersmith-Winograd tensor. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 1029–1046.

[13] Ken Goldberg, Theresa Roeder, Dhruv Gupta, and Chris Perkins. 2001. Eigentaste: A constant time collaborative filtering algorithm. *information retrieval* 4, 2 (2001), 133–151.

[14] Goetz Graefe and Harumi Kuno. 2011. Modern B-tree techniques. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 1370–1373.

[15] F Maxwell Harper and Joseph A Konstan. 2015. The movielens datasets: History and context. *Acm transactions on interactive intelligent systems (tiis)* 5, 4 (2015), 1–19.

[16] Axel Hertzschuch, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. 2021. Simplicity Done Right for Join Ordering. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org.

[17] VanPhi Ho and Dong-Joo Park. 2016. A survey of the-state-of-the-art b-tree index on flash memory. *International Journal of Software Engineering and Its Applications* 10, 4 (2016), 173–188.

[18] Xiao Hu and Ke Yi. 2020. Parallel Algorithms for Sparse Matrix Multiplication and Join-Aggregate Queries. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 411–425.

[19] Yu-Ching Hu, Yuliang Li, and Hung-Wei Tseng. 2021. TCUDB: Accelerating Database with Tensor Processors. *arXiv preprint arXiv:2112.07552* (2021).

[20] Zichun Huang and Shimin Chen. 2022. Density-optimized Intersection-free Mapping and Matrix Multiplication for Join-Project Operations (extended version). *arXiv e-prints arXiv:2206.04995*.

[21] Yannis E Ioannidis and Younkyung Cha Kang. 1991. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In *Proceedings of the 1991 ACM SIGMOD international conference on Management of data*. 168–177.

[22] Shunsuke Kanda, Kazuhiro Morita, and Masao Fuketa. 2017. Practical string dictionary compression using string dictionary encoding. In *2017 International Conference on Big Data Innovations and Applications (Innovate-Data)*. IEEE, 1–8.

[23] Jeremy Kepner and John Gilbert. 2011. *Graph algorithms in the language of linear algebra*. SIAM.

[24] Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1378–1389.

[25] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. 2013. Meet the walkers accelerating index traversals for in-memory databases. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 468–479.

[26] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*. 489–504.

[27] Jakub Kurzak, Wesley Alvaro, and Jack Dongarra. 2009. Optimizing matrix multiplication for a short-vector SIMD architecture–CELL processor. *Parallel Comput.* 35, 3 (2009), 138–150.

[28] Per-Åke Larson, Jonathan Goldstein, and Jingren Zhou. 2004. MTCache: Transparent Mid-Tier Database Caching in SQL Server. In *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA*. IEEE Computer Society, 177–188.

[29] Per-Åke Larson and H. Z. Yang. 1985. Computing Queries from Derived Relations. In *VLDB'85, Proceedings of 11th International Conference on Very Large Data Bases, August 21-23, 1985, Stockholm, Sweden*, Alain Pirotte and Yannis Vassiliou (Eds.). Morgan Kaufmann, 259–269.

[30] François Le Gall. 2012. Faster algorithms for rectangular matrix multiplication. In *2012 IEEE 53rd annual symposium on foundations of computer science*. IEEE, 514–523.

[31] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling.. In *Cidr*.

[32] Jure Leskovec, Lada A. Adamic, and Bernardo A. Huberman. 2007. The dynamics of viral marketing. *ACM Trans. Web* 1, 1 (2007), 5. https://doi.org/10.1145/1232722.1232727

[33] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. 2009. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics* 6, 1 (2009), 29–123.

[34] Priti Mishra and Margaret H Eich. 1992. Join processing in relational databases. *ACM Computing Surveys (CSUR)* 24, 1 (1992), 63–113.

[35] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the graph 500. *Cray Users Group (CUG)* 19 (2010), 45–74.

[36] Marco Pegoraro, Merih Seran Uysal, and Wil MP van der Aalst. 2020. Efficient time and space representation of uncertain event data. *Algorithms* 13, 11 (2020), 285.

[37] Yuan Qiu, Yilei Wang, Ke Yi, Feifei Li, Bin Wu, and Chaoqun Zhan. 2021. Weighted Distinct Sampling: Cardinality Estimation for SPJ Queries. In *Proceedings of the 2021 International Conference on Management of Data*. 1465–1477.

[38] Erik Saule, Kamer Kaya, and Ümit V Çatalyürek. 2013. Performance evaluation of sparse matrix multiplication kernels on intel xeon phi. In *International Conference on Parallel Processing and Applied Mathematics*. Springer, 559–570.

[39] Donovan A Schneider and David J DeWitt. 1990. *Tradeoffs in processing complex join queries via hashing in multiprocessor database machines*. University of Wisconsin-Madison. Computer Sciences Department.

[40] Malte Skarupke. 2017. I Wrote The Fastest Hashtable. https://probablydance.com/2017/02/26/i-wrote-the-fastest-hashtable/.

[41] Volker Strassen. 1969. Gaussian elimination is not optimal. *Numer. Math.* 13 (1969), 354–356.

[42] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing non-volatile memory in database systems. In *Proceedings of the 2018 International Conference on Management of Data*. 1541–1555.

[43] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 167–188.

[44] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems* 42, 1 (2015), 181–213.

[45] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems* 42, 1 (2015), 181–213.