I/O-Efficient Join Dependency Testing, Loomis-Whitney Join, and Triangle Enumeration[*]

Xiaocheng Hu
Chinese University of Hong Kong
xchu@cse.cuhk.edu.hk

Miao Qiao
Massey University
m.qiao@massey.ac.nz

Yufei Tao
University of Queensland
taoyf@itee.uq.edu.au

April 21, 2016

Abstract

We revisit two fundamental problems in database theory. The *join-dependency* (JD) *testing problem* is to determine whether a given JD holds on a relation r. We prove that the problem is NP-hard even if the JD involves only relations each of which has only two attributes. The *JD-existence testing problem* is to determine if there exists any non-trivial JD satisfied by r. We present an I/O-efficient algorithm in the external memory model, which in fact settles the closely related Loomis-Whitney enumeration problem. As a side product, we solve the triangle enumeration problem with the optimal I/O-complexity, improving a recent result of Pagh and Silvestri in PODS'14.

Keywords: Join Dependency, Loomis-Whitney Join, Triangle Enumeration, External Memory

[*]A preliminary version of this article appeared in PODS'15.

1 Introduction

Given a relation r of d attributes, a key question in database theory is to ask if r is *decomposable*, namely, whether r can be projected onto a set S of relations with less than d attributes such that the natural join of those relations equals precisely r. Intuitively, a *yes* answer to the question implies that r contains a certain form of redundancy. Some of the redundancy may be removed by decomposing r into the smaller (in terms of attribute number) relations in S, which can be joined together to restore r whenever needed. A *no* answer, on the other hand, implies that the decomposition of r based on S will lose information, as far as natural join is concerned.

Join Dependency Testing. The above question (as well as its variants) has been extensively studied by resorting to the notion of *join dependency* (JD). To formalize the notion, let us refer to d as the *arity* of r. Denote by $R = \{A_1, A_2, ..., A_d\}$ the set of names of the d attributes in r. R is called the *schema* of r. Sometimes we may denote r as $r(R)$ or $r(A_1, A_2, ..., A_d)$ to emphasize on its schema. Let $|r|$ represent the number of tuples in r.

A JD *defined on* R is an expression of the form

$$J = \bowtie[R_1, R_2, ..., R_m]$$

where (i) $m \geq 1$, (ii) each R_i ($1 \leq i \leq m$) is a subset of R that contains at least 2 attributes, and (iii) $\cup_{i=1}^{m} R_i = R$. J is *non-trivial* if none of $R_1, ..., R_m$ equals R. The *arity* of J is defined to be $\max_{i=1}^{m} |R_i|$, i.e., the largest size of $R_1, ..., R_m$. Clearly, the arity of a non-trivial J is between 2 and $d - 1$.

Relation r is said to *satisfy* J if

$$r = \pi_{R_1}(r) \bowtie \pi_{R_2}(r) \bowtie ... \bowtie \pi_{R_m}(r)$$

where $\pi_X(r)$ denotes the projection of r onto an attribute set X, and $\bowtie$ represents natural join. We are ready to formally state the first two problems studied in this article:

Problem 1. [λ-JD Testing] *Given a relation r and a join dependency J of arity at most λ that is defined on the schema of r, we want to determine whether r satisfies J.*

Problem 2. [JD Existence Testing] *Given a relation r, we want to determine whether there is any non-trivial join dependency J such that r satisfies J.*

Note the difference in the objectives of the above problems. Problem 1 aims to decide if r can be decomposed according to a *specific* set J of projections. On the other hand, Problem 2 aims to find out if there is any way to decompose r at all.

Computation Model. Our discussion on Problem 1 will concentrate on proving its NP-hardness. For this purpose, we will describe all our reductions in the standard RAM model.

For Problem 2, which is known to be polynomial time solvable (as we will explain shortly), the main issue is to design fast algorithms. We will do so in the *external memory* (EM) model [2], which has become the de facto model for analyzing I/O-efficient algorithms. Under this model, a machine is equipped with M words of memory, and an unbounded disk that has been formatted into *blocks* of B words. It holds that $M \geq 2B$. An *I/O operation* exchanges a block of data between the disk and the memory. The *cost* of an algorithm is defined to be the number of I/Os performed. CPU calculation is for free.

To avoid rounding, we define $\lg_x y = \max\{1, \log_x y\}$, and will describe all logarithms using $\lg_x y$. In all cases, the value of an attribute is assumed to fit in a single word.

Loomis-Whitney Enumeration. As will be clear later, the JD existence-testing problem is closely related to the so-called *Loomis-Whitney (LW) join* (this term was coined in [12]). Let $R = \{A_1, A_2, ..., A_d\}$ be a set of d attributes. For each $i \in [1, d]$, define $R_i = R \setminus \{A_i\}$, that is, removing A_i from R. Let $r_1, r_2, ..., r_d$ be d relations such that r_i $(1 \le i \le d)$ has schema R_i. Then, the natural join $r_1 \bowtie r_2 \bowtie ... \bowtie r_d$ is called an LW join. Note that the schema of the join result is R.

We will consider LW joins in the EM model, where traditionally a join must write out all the tuples in the result to the disk. However, the result size can be so huge that the number of I/Os for writing the result may (by far) overwhelm the cost of the join's rest execution. Furthermore, in some applications of LW joins (e.g., for solving Problem 2), it is not necessary to actually write the result tuples to the disk; instead, it suffices to *witness* each result tuple once in the memory.

Because of the above, we follow the approach of [14] by studying an *enumerate* version of the problem. Specifically, we are given a memory-resident routine *emit*(.) which requires $O(1)$ words to store. The parameter of the routine is a tuple t of d values $(a_1, ..., a_d)$ such that a_i is in the domain of A_i for each $i \in [1, d]$. The routine simply sends out t to an outbound socket with no I/O cost. Then, our problem can be formally stated as:

Problem 3. [LW Enumeration] *Given relations $r_1, ..., r_d$ as defined earlier where $d \le M/2$, we want to invoke emit(t) once and exactly once for each tuple $t \in r_1 \bowtie r_2 \bowtie ... \bowtie r_d$.*

As a noteworthy remark, if an algorithm can solve the above problem in x I/Os using $M - B$ words of memory, then it can also report the entire LW join result of K tuples (i.e., totally Kd values) in $x + O(Kd/B)$ I/Os.

Triangle Enumeration. Besides being a stepping stone for Problem 2, LW enumeration has relevance to several other problems, among which the most prominent one is perhaps the *triangle enumeration problem* [14] due to its large variety of applications (see [8, 14] and the references therein for an extensive summary).

Let $G = (V, E)$ be an undirected simple graph, where V (or E) is the set of vertices (or edges, resp.). A *triangle* is defined as a clique of 3 vertices in G. We are again given a memory-resident routine *emit*(.) that occupies $O(1)$ words. This time, given a triangle Δ as its parameter, the routine sends out Δ to an outbound socket with no I/O cost (this implies that all the 3 edges of Δ must be in the memory at this moment). Then, the triangle enumeration problem can be formally stated as:

Problem 4. [Triangle Enumeration] *Given graph G as defined earlier, we want to invoke emit(Δ) once and exactly once for each triangle Δ in G.*

Observe that this is merely a special instance of LW enumeration with $d = 3$ where $r_1 = r_2 = r_3 = E$ (specifically, E is regarded as a relation with two columns, such that every edge (u, v) gives rise to two tuples (u, v) and (v, u) in the relation), with some straightforward care to avoid emitting a triangle twice in no extra I/O cost.

1.1 Previous Results

Join Dependency Testing. Beeri and Vardi [5] proved that λ-JD testing (Problem 1) is NP-hard if $\lambda = d - o(d)$; recall that d is the number of attributes in the input relation r. Maier, Sagiv, and Yannakakis [11] gave a stronger proof showing that λ-JD testing is still NP-hard for $\lambda = \Omega(d)$

(more specifically, roughly $2d/3$). In other words, (unless P = NP) no polynomial-time algorithm can exist to verify every JD $\bowtie[R_1, R_2, ..., R_m]$ on r, when one of $R_1, ..., R_m$ has $\Omega(d)$ attributes.

However, the above result does not rule out the possibility of efficient testing when the JD has a small arity, namely, *all* of $R_1, ..., R_m$ have just a few attributes (e.g., as few as just 2). Small-arity JDs are important because many relations in reality can eventually be losslessly decomposed into relations with small arities. By definition, for any $\lambda_1 < \lambda_2$, the λ_1-JD testing problem may only be easier than λ_2-JD testing problem because an algorithm for the latter can be used to solve the former problem, but not the vice versa. The ultimate question, therefore, is whether 2-JD testing can be solved within polynomial time. Unfortunately, the arity of J being $\Omega(d)$ appears to be an inherent requirement in the reductions of [5, 11].

We note that a large body of beautiful theory has been developed on *dependency inference*, where the objective is to determine whether a target dependency can be inferred from a set Σ of dependencies (see [1, 10] for excellent guides into the literature). When the target dependency is a join dependency, the inference problem has been proven to be NP-hard in a variety of scenarios, most notably: (i) when Σ contains one join dependency and a set of functional dependencies [5, 11], (ii) when Σ is a set of multi-valued dependencies [6], and (iii) when Σ has one domain dependency and a set of functional dependencies [9]. The proofs of [5, 11] are essentially the same ones used to establish the NP-hardness of $\Omega(d)$-JD testing, while those of [6, 9] do not imply any conclusions on λ-JD testing.

JD Existence Testing and LW Join. There is an interesting connection between JD existence testing (Problem 2) and LW join. Let $r(R)$ be the input relation to Problem 2, where $R = \{A_1, A_2, ..., A_d\}$. For each $i \in [1, d]$, define $R_i = R \setminus \{A_i\}$, and $r_i = \pi_{R_i}(r)$. Nicolas showed [13] that r satisfies at least one non-trivial JD *if and only if* $r = r_1 \bowtie r_2 \bowtie ... \bowtie r_d$. In fact, since it is always true that $r \subseteq r_1 \bowtie r_2 \bowtie ... \bowtie r_d$, Problem 2 has an answer *yes* if and only if $r_1 \bowtie r_2 \bowtie ... \bowtie r_d$ returns exactly $|r|$ result tuples.

Therefore, Problem 2 boils down to evaluating the result size of the LW join $r_1 \bowtie r_2 \bowtie ... \bowtie r_d$. Atserias, Grohe, and Marx [4] showed that the result size can be as large as $(n_1 n_2 ... n_d)^{\frac{1}{d-1}}$, where $n_i = |r_i|$ for each $i \in [1, d]$. They also gave a RAM algorithm to compute the join result in $O(d^2 \cdot (n_1 n_2 ... n_d)^{\frac{1}{d-1}} \cdot \sum_{i=1}^{n} n_i)$ time. Since apparently $n_i \le n = |r|$ ($1 \le i \le d$), it follows that their algorithm has running time $O(d^2 \cdot n^{d/(d-1)} \cdot dn) = O(d^3 \cdot n^{2+o(1)})$, which in turn means that Problem 2 is solvable in polynomial time. Ngo et al. [12] designed a faster RAM algorithm to perform the LW join (hence, solving Problem 2) in $O(d^2 \cdot (n_1 n_2 ... n_d)^{\frac{1}{d-1}} + d^2 \sum_{i=1}^{d} n_i)$ time. For constant d, Veldulzen [15] presented a simplified algorithm to achieve the same complexity.

Problems 2 and 3 become much more challenging in external memory (EM). All the algorithms of [4, 12, 15] rely on the fact that *dictionary search*, i.e., finding a specific element in a set, can be done efficiently in RAM (either in $O(1)$ expected time or logarithmic worse-case time). In EM, however, spending even just one I/O per for dictionary search is excessively expensive for a problem like LW join because the number of such searches is huge. Because of this, when adapted to EM, the algorithms of [4, 12, 15] do not offer competitive efficiency; for instance, that of [12] can entail up to $O(d^2 \cdot (n_1 n_2 ... n_d)^{\frac{1}{d-1}} + d^2 \sum_{i=1}^{d} n_i)$ I/Os. When d is small, this may be even worse than a naive generalized blocked-nested loop, whose I/O complexity for $d = O(1)$ is $O(n_1 n_2 ... n_d / (M^{d-1} B))$ I/Os. Recall that B and M are the sizes of a disk block and memory, respectively.

Triangle Enumeration. Problem 4 has received a large amount of attention from the database and theory communities (see [8] for a survey). Recently, Pagh and Silvestri [14] solved the problem in EM with a randomized algorithm whose I/O cost is $O(|E|^{1.5}/(\sqrt{M}B))$ *expected*, where $|E|$ is the number of edges in the input graph. They also presented a sophisticated de-randomization technique to convert their algorithm into a deterministic one that performs $O(\frac{|E|^{1.5}}{\sqrt{M}B} \cdot \log_{M/B} \frac{|E|}{B})$ I/Os. An I/O lower bound of $\Omega(|E|^{1.5}/(\sqrt{M}B))$ has been independently developed in [8, 14] on the *witnessing* class of algorithms.

1.2 Our Results

Our first main result is:

Theorem 1. *2-JD testing is NP-hard.*

The theorem officially puts a negative answer to the question whether a small-arity JD can be tested efficiently (remember that 2 is already the smallest possible arity). As a consequence, we know that Problem 2 is NP-hard for *every* value $\lambda \in [2, d-1]$. Our proof is completely different from those of [5, 11], and is based on a novel reduction from the *Hamiltonian path problem.*

Our second main result is an I/O-efficient algorithm for LW enumeration (Problem 3). Let $r_1, r_2, ..., r_d$ be the input relations; and set $n_i = |r_i|$. We will prove:

Theorem 2. *There is an EM algorithm that solves the LW enumeration problem with I/O complexity:*

$$O\Big(sort\Big[d^{3+o(1)}\Big(\frac{\Pi_{i=1}^{d} n_i}{M}\Big)^{\frac{1}{d-1}} + d^2 \sum_{i=1}^{d} n_i\Big]\Big).$$

where function $sort(x) = (x/B)\lg_{M/B}(x/B)$.

The main obstacle we faced in performing LW enumeration I/O-efficiently is that, we can no longer rely on repetitive dictionary search, as is a key component of all the RAM algorithms [4, 12, 15]. As mentioned in Section 1.1, while performing a dictionary search in $O(1)$ time is good in RAM, it is prohibitively expensive to spend $O(1)$ I/Os for the same purpose in EM. To overcome the obstacle, we abandoned dictionary search completely; in fact, our major contribution is a delicate piece of recursive machinery, which resorts to only sorting and scanning.

As our third main result, we prove in Section 4 an improved version of Theorem 2 for $d = 3$:

Theorem 3. *There is an EM algorithm that solves the LW enumeration problem of* $d = 3$ *with I/O complexity* $O(\frac{1}{B}\sqrt{\frac{n_1 n_2 n_3}{M}} + sort(n_1 + n_2 + n_3))$.

By combining the above two theorems with the reduction from JD existence testing to LW enumeration described in Section 1.1, we obtain the first non-trivial algorithm for I/O-efficient JD existence testing (Problem 2):

Corollary 1. *Let* $r(R)$ *be the input relation to the JD existence testing problem, where* $R = \{A_1, ..., A_d\}$. *For each* $i \in [1, d]$, *define* $R_i = R \setminus \{A_i\}$, *and* n_i *as the number of tuples in* $\pi_{R_i}(r)$. *Then:*

- *For* $d > 3$, *the problem can be solved with the I/O complexity in Theorem 2.*
- *For* $d = 3$, *the I/O complexity can be improved to the one in Theorem 3.*

Finally, when $n_1 = n_2 = n_3 = |E|$, Theorem 3 directly gives a new algorithm for triangle enumeration (Problem 4), noticing that $sort(|E|) = O(|E|^{1.5}/(\sqrt{M}B))$:

Corollary 2. *There is an algorithm that solves the triangle enumeration problem optimally in* $O(|E|^{1.5}/(\sqrt{M}B))$ *I/Os.*

Our triangle enumeration algorithm is deterministic, and strictly improves that of [14] by a factor of $O(\lg_{M/B}(|E|/B))$. Furthermore, the algorithm belongs to the witnessing class [8], and is the first (deterministic algorithm) in this class achieving the optimal I/O complexity for all values of M and B.

2 NP-Hardness of 2-JD Testing

This section will establish Theorem 1 with a reduction from the *Hamiltonian path problem.* Let $G = (V, E)$ be an undirected simple graph (a graph is *simple* if it has at most one edge between any two vertices). with a vertex set V and an edge set E. Set $n = |V|$ and $m = |E|$. A *path* of length ℓ in G is a sequence of ℓ vertices $v_1, v_2, ..., v_\ell$ such that E has an edge between v_i and v_{i+1} for each $i \in [1, \ell - 1]$. The path is *simple* if no two vertices in the path are the same. A *Hamiltonian path* is a simple path in G of length n (such a path must pass each vertex in V exactly once). Deciding whether G has a Hamiltonian path is known to be NP-hard [7].

Let R be a set of n attributes: $\{A_1, A_2, ..., A_n\}$. We will create $\binom{n}{2}$ relations. Specifically, for each pair of i, j such that $1 \le i < j \le n$, we generate a relation $r_{i,j}$ with attributes A_i, A_j. The tuples in $r_{i,j}$ are determined as follows:

- *Case* $j = i + 1$: Initially, $r_{i,j}$ is empty. For each edge E between vertices u and v, we add two tuples to $r_{i,j}$: (u, v) and (v, u). In total, $r_{i,j}$ has $2m$ tuples.
- *Case* $j \ge i + 2$: $r_{i,j}$ contains $n(n - 1)$ tuples (x, y), for all possible integers x, y such that $x \ne y$, and $1 \le x, y \le n$.

The total number of tuples in the $r_{i,j}$ of all possible i, j is $O(nm + n^4) = O(n^4)$. Define:

$$\text{CLIQUE} \quad = \quad \text{the output of the natural join of all } r_{i,j} (1 \le i < j \le n).$$

As an example, for $n = 3$, $\text{CLIQUE} = r_{1,2} \bowtie r_{1,3} \bowtie r_{2,3}$. In general, CLIQUE is a relation with schema R. It should be easy to observe the fact below:

Proposition 1. *G has a Hamiltonian path if and only if* CLIQUE *is not empty.*

For each pair of i, j satisfying $1 \le i < j \le n$, define an attribute set $R_{i,j} = \{A_i, A_j\}$. Denote by J the JD that "corresponds to" CLIQUE, namely:

$$J \quad = \quad \bowtie[R_{i,j}, \forall i, j \text{ s.t. } 1 \le i < j \le n].$$

For instance, for $n = 3$, $J = \bowtie[R_{1,2},\ R_{1,3},\ R_{2,3}]$. Note that J has arity 2, and $R = \cup_{i,j} R_{i,j}$ in general.

Next, we will construct from G a relation r^* of schema R such that CLIQUE is empty *if and only if* r^* satisfies J. Initially, r^* is empty. For *every* tuple t in *every* relation $r_{i,j}$ $(1 \le i < j \le n)$, we will insert a tuple t' into r^*. Recall that $r_{i,j}$ has schema $\{A_i, A_j\}$. Suppose, without loss of generality, that $t = (a_i, a_j)$. Then, t' is determined as follows:

- $t'[A_i] = a_i$ ($t'[A_i]$ is the value of t' on attribute A_i)
- $t'[A_j] = a_j$

- For any $k \in [1, n]$ but $k \neq i$ and $k \neq j$, $t'[A_k]$ is set to a *dummy value* that appears only once in the whole r^*.

Since there are $O(n^4)$ tuples in the $r_{i,j}$ of all i, j, we know that r^* has $O(n^4)$ tuples, and hence, can be built in $O(n^5)$ time.

Lemma 1. Clique *is empty if and only if r^* satisfies J.*

Proof. We first point out three facts:

1. Every tuple in r^* has $n - 2$ dummy values.

2. Define $r^*_{i,j} = \pi_{A_i,A_j}(r^*)$ for i, j satisfying $1 \leq i < j \leq n$. Clearly, $r^*_{i,j}$ and $r_{i,j}$ share the same schema $R_{i,j}$. It is easy to verify that $r_{i,j}$ is exactly the set of tuples in $r^*_{i,j}$ that do *not* contain dummy values.

3. Define:

$$\text{Clique}^* \quad = \quad \text{the output of the natural join of all } r^*_{i,j} \ (1 \leq i < j \leq n).$$

   Then, r^* satisfies J if and only if $r^* = \text{Clique}^*$.

Equipped with these facts, we now proceed to prove the lemma.

For the "if" direction, assuming that r^* satisfies J, we need to show that Clique is empty. Suppose, on the contrary, that $(a_1, a_2, ..., a_n)$ is a tuple in Clique. Hence, (a_i, a_j) is a tuple in $r_{i,j}$ for any i, j satisfying $1 \leq i < j \leq n$. As neither a_i nor a_j is dummy, by Fact 2, we know that (a_i, a_j) belongs to $r^*_{i,j}$. It thus follows that $(a_1, a_2, ..., a_n)$ is a tuple in Clique*. However, by Fact 1, $(a_1, a_2, ..., a_n)$ cannot belong to r^*, thus giving a contradiction against Fact 3.

For the "only-if" direction, assuming that Clique is empty, we need to show that r^* satisfies J. Suppose, on the contrary, that r^* does not satisfy J, namely, $r^* \neq \text{Clique}^*$ (Fact 3). Let $(a^*_1, a^*_2, ..., a^*_n)$ be a tuple in Clique* but not in r^*. We distinguish two cases:

- *Case 1: none of $a^*_1, ..., a^*_n$ is dummy.* This means that, for any i, j satisfying $1 \leq i < j \leq n$, (a^*_i, a^*_j) is a tuple in $r_{i,j}$ (Fact 2). Therefore, $(a^*_1, a^*_2, ..., a^*_n)$ must be a tuple in Clique, contradicting the assumption that Clique is empty.

- *Case 2: a^*_k is dummy for at least one $k \in [1, n]$.* Since every dummy value appears exactly once in r^*, we can identify a unique tuple t^* in r^* such that $t^*[A_k] = a^*_k$. Next, we will show that t^* is precisely $(a^*_1, a^*_2, ..., a^*_n)$, thus contradicting the assumption that $(a^*_1, a^*_2, ..., a^*_n)$ is not in r^*, which will then complete the proof.

  Consider any i such that $1 \leq i < k$. That $(a^*_1, a^*_2, ..., a^*_n)$ is in Clique* implies that (a^*_i, a^*_k) is in $r^*_{i,k}$. However, because in r^* the value a^*_k appears only in t^*, it must hold that $t^*[A_i] = a^*_i$. By a similar argument, for any j such that $k < j \leq n$, we must have $t^*[A_j] = a^*_j$. It thus follows that $(a^*_1, a^*_2, ..., a^*_n)$ is precisely t^*.

□

From the above discussion, we know that any 2-JD testing algorithm can be used to check whether Clique is empty (Lemma 1), and hence, can be used to check whether G has a Hamiltonian path (Lemma 1). We thus conclude that 2-JD testing is NP-hard.

3 LW Enumeration

The discussion from the previous section has eliminated the hope of efficient JD testing no matter how small the JD arity is (unless P = NP). We therefore switch to the less stringent goal of JD *existence* testing (Problem 2). Based on the reduction described in Section 1.1, next we concentrate on LW enumeration as formulated in Problem 3, and will establish Theorem 2.

Let us recall a few basic definitions. We have a "global" set of attributes $R = \{A_1, A_2, ..., A_d\}$. For each $i \in [1, d]$, let $R_i = R \setminus \{A_i\}$. We are given relations $r_1, r_2, ..., r_d$ where r_i $(1 \le i \le d)$ has schema R_i. The objective of LW enumeration is that, for every tuple t in the result of $r_1 \bowtie r_2 \bowtie ... \bowtie r_d$, we should invoke $emit(t)$ once and exactly once. We want to do so I/O-efficiently in the EM model, where B and M represent the sizes (in words) of a disk block and memory, respectively.

For each $i \in [1, d]$, set $n_i = |r_i|$, and define $dom(A_i)$ as the domain of attribute A_i. Given a tuple t and an attribute A_i (in the schema of the relation containing t), we denote by $t[A_i]$ the value of t on A_i. Furthermore, we assume that each of $r_1, ..., r_d$ is given in an array, but the d arrays do not need to be consecutive.

3.1 Basic Algorithms

Let us first deal with two scenarios under which LW enumeration is easier.

3.1.1 Small Join

The first scenario arises when there is an n_i (for some $i \in [1, d]$) satisfying $n_i = O(M/d)$. In such a case, we call $r_1 \bowtie r_2 \bowtie ... \bowtie r_d$ a *small join*. Next, we prove:

Lemma 2. *Given a small join, we can emit all its result tuples in $O(d + sort(d \sum_{i=1}^{d} n_i))$ I/Os.*

Without loss of generality, suppose that r_1 has the smallest cardinality among all the input relations. Let us first assume that $n_1 \le cM/d$ where c is a sufficiently small constant so that r_1 can be kept in memory throughout the entire algorithm. With r_1 already in memory, we merge all the tuples of $r_2, ..., r_d$ into a set L, sorted by attribute A_1. For each $a \in dom(A_1)$, let $L[a]$ be the set of tuples in L whose A_1-values equal a.

Next, for each $a \in dom(A_1)$, we use the procedure below to emit all such tuples $t^* \in r_1 \bowtie r_2 \bowtie ... \bowtie r_d$ that $t^*[A_1] = a$. First, initialize empty sets $S_2, ..., S_d$ in memory. Then, process each tuple $t \in L[a]$ as follows. Suppose that t originates from r_i for some $i \in [2, d]$. Check whether r_1 has a tuple t' satisfying

$$t'[A_j] = t[A_j], \qquad \forall j \in [2, d] \setminus \{i\}. \tag{1}$$

If the answer is no, t is discarded; otherwise, we add it to S_i. Note that the checking happens in memory and entails no I/O. Having processed all the tuples of $L[a]$ this way, we emit all the tuples in the result of $r_1 \bowtie S_2 \bowtie S_3 \bowtie ... \bowtie S_d$ (these are exactly the tuples in $r_1 \bowtie r_2 \bowtie ... \bowtie r_d$ whose A_1-values equal a). The above tuple emission incurs no I/Os due to the following lemma.

Lemma 3. *$r_1, S_2, ..., S_d$ fit in memory.*

Proof. It is easy to show that $|S_i| \le n_1 \le cM/d$ for each $i \in [2, d]$. A naive way to store S_i takes $d|S_i|$ words, in which case we would need $\Omega(dM)$ words to store $r_1, S_2, ..., S_d$, exceeding the memory capacity M. To remedy this issue, we store S_i using only $|S_i|$ words as follows. Given a tuple $t \in S_i$, we store a single integer that is the memory address of the tuple t' in (1), which requires only $\lg_2 n_1$

bits by storing an offset. This does not lose any information because we can recover t by resorting to (1) and the fact that $t[A_1] = a$. Therefore, $r_1, S_2, ..., S_d$ can be represented in $O(d \cdot n_1)$ words, which is smaller than M when the constant c is sufficiently small. □

The overall cost of the algorithm is dominated by the cost of (i) merging $r_2, ..., r_d$ into L, which takes $O(d + (d/B) \sum_{i=2}^{d} n_i)$ I/Os, and (ii) sorting L, which takes $O(sort(d \sum_{i=2}^{d} n_i))$ I/Os, using an algorithm of [3] (see Theorem 1 of [3]) for string sorting in EM. Hence, the overall I/O complexity is as claimed in Lemma 2.

For the case where $n_1 > cM/d$, we simply divide r_1 arbitrarily into $O(1)$ subsets each with cM/d tuples, and then apply the above algorithm to emit all the result tuples produced from each of the subsets. This concludes the proof of Lemma 2.

3.1.2 Point Join

The second scenario where LW enumeration is relatively easy takes a bit more efforts to explain. In addition to $r_1, ..., r_d$, we accept two more input parameters:

- an integer $H \in [1, d]$
- a value $a \in dom(A_H)$.

It is required that a should be the *only* value that appears in the A_H attributes of $r_1, ..., r_{H-1}, r_{H+1}, ..., r_d$ (recall that r_H does not have A_H). In such a case, we call $r_1 \bowtie r_2 \bowtie ... \bowtie r_d$ a *point join*. Next, we prove:

Lemma 4. *Given a point join, we can emit all its result tuples in* $O(d + sort(d^2 n_H + d \sum_{i \in [1,d] \setminus \{H\}} n_i))$ *I/Os.*

Proof. For each $i \in [1, d] \setminus \{H\}$, define $X_i = R_i \cap R_H$ (i.e., X_i includes all the attributes in R except A_i and A_H).

In ascending order of $i \in [1, d] \setminus \{H\}$, we invoke the procedure below to process r_i and r_H, which continuously removes some tuples from r_H. First, sort r_i and r_H by X_i, respectively. Then, synchronously scan r_i and r_H according to the sorted order. For each tuple t in r_H, we check during the scan whether r_i has a tuple t' that has the same values as t on *all* the attributes in X_i—note that such t' (if exists) must be unique, due to the fact that a is the only A_H value in r_i. The sorted order ensures that if t' exists, then t and t' must appear consecutively during the synchronous scan. If t' exists, t is kept in r_H; otherwise, we discard t from r_H (t cannot produce any tuple in $r_1 \bowtie r_2 \bowtie ... \bowtie r_d$).

After the above procedure has finished through all $i \in [1, d] \setminus \{H\}$, we know that every tuple t remaining in r_H *must* produce exactly one result tuple t'' in $r_1 \bowtie r_2 \bowtie ... \bowtie r_d$ where $t''[A_i] = t[A_i]$ for all $i \in [1, d] \setminus \{H\}$, and $t'[A_H] = a$. Therefore, we can emit all such t' with one more scan of the (current) r_H.

The claimed I/O cost follows from the fact that r_H is sorted $d - 1$ times in total, while r_i is sorted once for each $i \in [1, d] \setminus \{H\}$. □

In what follows, we will denote the algorithm in the above lemma as PtJoin$(H, a, r_1, r_2, ..., r_d)$.

3.2 The Full Algorithm

This subsection presents an algorithm for solving the general LW enumeration problem. If $n_1 \le 2M/d$, we solve the problem directly by Lemma 2. The subsequent discussion focuses on $n_1 > 2M/d$.

Define:

$$U = \left(\frac{\prod_{i=1}^{d} n_i}{M} \right)^{\frac{1}{d-1}} \tag{2}$$

$$\tau_i = \frac{n_1 n_2 ... n_i}{(U \cdot d^{\frac{1}{d-1}})^{i-1}} \quad \text{for each } i \in [1, d]. \tag{3}$$

Notice that $\tau_1 = n_1$ and $\tau_d = M/d$.

Our algorithm is a recursive procedure $\textsc{Join}(h, \rho_1, ..., \rho_d)$, which has three requirements:

- h is an integer in $[1, d]$;
- Each ρ_i $(1 \le i \le d)$ is a subset of the tuples in r_i.
- The size of ρ_1 satisfies:

$$|\rho_1| \le \tau_h. \tag{4}$$

$\textsc{Join}(h, \rho_1, ..., \rho_d)$ emits all result tuples in $\rho_1 \bowtie ... \bowtie \rho_d$. The LW enumeration problem can be settled by calling $\textsc{Join}(1, r_1, ..., r_d)$.

If $\tau_h \le 2M/d$, $|\rho_1| \le \tau_h = O(M/d)$; $\textsc{Join}(h, \rho_1, ..., \rho_d)$ simply runs the small-join algorithm of Lemma 2. Next, we focus on $\tau_h > 2M/d$. Define:

$$H = \min\{i \in [h+1, d] \mid \tau_i < \tau_h/2\}. \tag{5}$$

H always exists because $\tau_d = M/d < \tau_h/2$. Given a value $a \in dom(A_H)$, define

$$freq(a) = |\{t \in \rho_1 \mid t[A_H] = a\}|.$$

We collect all the frequent values a into a set Φ:

$$\Phi = \{a \in dom(A_H) \mid freq(a) > \tau_H/2\}. \tag{6}$$

For each $i \in [1, d] \setminus \{H\}$, we partition ρ_i into:

$$\rho_i^{heavy} = \{t \in \rho_i \mid t[A_H] \in \Phi\}$$
$$\rho_i^{light} = \{t \in \rho_i \mid t[A_H] \notin \Phi\}$$

It is rudimentary to produce Φ, as well as ρ_i^{heavy} and ρ_i^{light} for each $i \in [1, d] \setminus \{H\}$, by sorting on A_H. Specifically, each element to be sorted is a tuple of $d - 1$ values where $d \le M/2$ (see the definition of Problem 3). Using an EM string sorting algorithm of [3], all the sorting can be completed with $O(d + sort(d \sum_{i \in [1,d] \setminus \{H\}} |\rho_i|))$ I/Os in total.

A result tuple $t^* \in \rho_1 \bowtie ... \bowtie \rho_d$ is said to be *heavy* if $t^*[A_H] \in \Phi$, or *light* otherwise. The set of heavy tuples is precisely

$$\rho_1^{heavy} \bowtie ... \bowtie \rho_{H-1}^{heavy} \bowtie \rho_H \bowtie \rho_{H+1}^{heavy} \bowtie ... \bowtie \rho_d^{heavy},$$

whereas the set of light tuples is

$$\rho_1^{light} \bowtie ... \bowtie \rho_{H-1}^{light} \bowtie \rho_H \bowtie \rho_{H+1}^{light} \bowtie ... \bowtie \rho_d^{light}.$$

We will emit heavy and light tuples separately, as explained next.

3.2.1 Enumerating Heavy Tuples

For every $a \in \Phi$, define for each $i \in [1, d] \setminus \{H\}$:

$$\rho_i^{heavy}[a] \quad = \quad \{t \in \rho_i^{heavy} \mid t[A_H] = a\}.$$

The tuples of $\rho_i^{heavy}[a]$ are stored consecutively in the disk because we have sorted ρ_i^{heavy} by A_H earlier. Hence, all the heavy tuples t^* with $t^*[A_H] = a$ can be emitted by a point join:

$$\textsc{PtJoin}(H, a, \rho_1^{heavy}[a], ..., \rho_{H-1}^{heavy}[a], \rho_H, \rho_{H+1}^{heavy}[a], ..., \rho_d^{heavy}[a]).$$

3.2.2 Enumerating Light Tuples

For each $i \in [1, d] \setminus \{H\}$, given an interval I in $dom(A_H)$, define:

$$\rho_i^{light}[I] \quad = \quad \{t \in \rho_i^{light} \mid t[A_H] \in I\}.$$

With one scan of ρ_1 (which has been sorted by A_H), we can obtain a sequence of disjoint intervals $I_1, I_2, ..., I_q$ with the properties below:

- $q = O(1 + |\rho_1|/\tau_H)$;
- $I_1, I_2, ..., I_q$ are in ascending order, and constitute a partition of $dom(A_H)$;
- The following *balancing condition* is fulfilled:
  - $|\rho_1^{light}[I_j]| \in [\tau_H/2, \tau_H]$ for every $j \in [1, q-1]$;
  - $|\rho_1^{light}[I_q]| \in [1, \tau_H]$.

Next, for all $i \in [2, d] \setminus \{H\}$, we produce $\rho_i^{light}[I_1]$, $\rho_i^{light}[I_2]$, ..., $\rho_i^{light}[I_q]$ by sorting. To emit all the light tuples, we recursively invoke our algorithm for each $j \in [1, q]$:

$$\textsc{Join}(H, \rho_1^{light}[I_j], ..., \rho_{H-1}^{light}[I_j], \rho_H, \rho_{H+1}^{light}[I_j], ..., \rho_d^{light}[I_j]).$$

This completes the description of our LW enumeration algorithm.

We remark that a crucial design of our recursive machinery is in how the parameter h in $\textsc{Join}(h, \rho_1, ..., \rho_d)$ increases. Specficailly, this parameter does not increase by 1 each time; rather, it will be set to the value of H as given by (5), which is at least $h + 1$ but can be larger.

3.3 A Recurrence on the I/O Cost

This and the next subsections will analyze the performance of our algorithm. Define a sequence of integers as follows:

- $h_1 = 1$;
- Provided that h_i ($i \geq 1$) has been defined:
  - if $\tau_{h_i} > 2M/d$, define $h_{i+1} = \min\{j \in [1 + h_i, d] \mid \tau_j < \tau_{h_i}/2\}$;
  - otherwise, h_{i+1} is undefined.

Denote by w the largest integer j such that h_j is defined.

Recall that our LW enumeration algorithm starts by calling the JOIN procedure with JOIN$(1, r_1, ..., r_d)$, which recursively makes subsequent calls to the same procedure. These calls form a tree $\mathcal{T}$. Equipped with the sequence $h_1, h_2, ..., h_w$, we can describe $\mathcal{T}$ in a more specific manner. Given a call JOIN$(h, \rho_1, ..., \rho_d)$, let us refer to the value of h as the call's *axis*. The initial call JOIN$(1, r_1, ..., r_d)$ has axis $h_1 = 1$. In general, an axis-h_i ($i \in [1, w-1]$) call generates axis-h_{i+1} calls, and hence, *parents* those calls in $\mathcal{T}$. Finally, all axis-h_w calls are leaf nodes in $\mathcal{T}$ (recall that an axis-h_w call simply invokes the small-join algorithm of Lemma 2). In other words, $\mathcal{T}$ has w *levels*; and all the calls at level $\ell \in [1, w]$ have an identical axis h_ℓ.

Given a level $\ell \in [1, w]$, define function $cost(\ell, \rho_1, ..., \rho_d)$ to be the number of I/Os performed by JOIN$(h_\ell, \rho_1, ..., \rho_d)$. We will work out a recurrence on $cost(\ell, \rho_1, ..., \rho_d)$, and then concentrate on solving it.

The Base Case of the Recurrence. If $\ell = w$, Lemma 2 immediately shows

$$cost(\ell, \rho_1, ..., \rho_d) \quad = \quad O\Big(d + sort\Big(d \sum_{i=1}^{d} |\rho_i|\Big)\Big). \tag{7}$$

The General Case. For $\ell \in [1, w-1]$, define:

$$\mu_\ell \quad = \quad \frac{2\tau_{h_\ell}}{\tau_{h_{\ell+1}}}. \tag{8}$$

By the way $h_{\ell+1}$ is selected, it must hold that $\mu_\ell \geq 4$. Furthermore, we can prove:

Lemma 5. *$\mu_{\ell-1} = O(Ud^{\frac{1}{d-1}}/n_{h_\ell})$ for each $\ell \in [2, w]$.*

Proof. It suffices to show that $\tau_{h_{\ell-1}}/\tau_{h_\ell} = O(Ud^{\frac{1}{d-1}}/n_{h_\ell})$. From (3), we get

$$\frac{\tau_{h_{\ell-1}}}{\tau_{h_\ell}} = \frac{(Ud^{\frac{1}{d-1}})^{h_\ell - h_{\ell-1}}}{\prod_{j=1+h_{\ell-1}}^{h_\ell} n_j}. \tag{9}$$

If $h_\ell = 1 + h_{\ell-1}$, then

$$(9) = \frac{Ud^{\frac{1}{d-1}}}{n_{h_\ell}}.$$

Otherwise ($h_\ell > 1 + h_{\ell-1}$), the definition of h_ℓ indicates $\tau_{h_\ell - 1} \geq \tau_{h_{\ell-1}}/2$ (otherwise, h_ℓ would not be the smallest integer $j \in [1 + h_{\ell-1}, d]$ satisfying $\tau_j < \tau_{h_{\ell-1}}/2$), namely:

$$\frac{\tau_{h_{\ell-1}}}{\tau_{h_\ell - 1}} = \frac{(Ud^{\frac{1}{d-1}})^{h_\ell - 1 - h_{\ell-1}}}{\prod_{j=1+h_{\ell-1}}^{h_\ell - 1} n_j} \leq 2.$$

Hence,

$$(9) = \frac{(Ud^{\frac{1}{d-1}})^{h_\ell - 1 - h_{\ell-1}}}{\prod_{j=1+h_{\ell-1}}^{h_\ell - 1} n_j} \cdot \frac{Ud^{\frac{1}{d-1}}}{n_{h_\ell}} \leq 2 \cdot \frac{Ud^{\frac{1}{d-1}}}{n_{h_\ell}}.$$

$\square$

Consider the set Φ defined in (6). Recall that for every $a \in \Phi$, $\mathit{freq}(a) > \tau_{h_{\ell+1}}/2$. Hence:

$$|\Phi| < \frac{|\rho_1|}{\tau_{h_{\ell+1}}/2} \le 2\frac{\tau_{h_\ell}}{\tau_{h_{\ell+1}}} = \mu_\ell.$$

where the second inequality is due to (4).

The number of I/Os that $\textsc{Join}(h_\ell, \rho_1, ..., \rho_d)$ spends on emitting heavy tuples is dominated by that of the point joins performed, whose total I/O cost (by Lemma 4) is:

$$\begin{aligned}
& O\Big(\sum_{a \in \Phi} \Big(d + \mathit{sort}\Big(d^2 \cdot |\rho_{h_{\ell+1}}| + d \sum_{i \in [1,d] \setminus \{h_{\ell+1}\}} \Big|\rho_i^{\mathit{heavy}}[a]\Big|\Big)\Big)\Big) \\
= \;& O\Big(d \cdot |\Phi| + \mathit{sort}\Big(d^2 \cdot |\Phi| \cdot |\rho_{h_{\ell+1}}| + d \sum_{i=1}^{d} |\rho_i|\Big)\Big) \\
= \;& O\Big(d \cdot \mu_\ell + \mathit{sort}\Big(d^2 \cdot \mu_\ell \cdot |\rho_{h_{\ell+1}}| + d \sum_{i=1}^{d} |\rho_i|\Big)\Big). \qquad (10)
\end{aligned}$$

The cost of emitting light tuples comes from recursion. Taking into account the fact that the sorting cost in Section 3.2.2 has been absorbed in (10), we have

$$\begin{aligned}
& \mathit{cost}(\ell, \rho_1, ..., \rho_d) \\
= \;& (10) + \sum_{j=1}^{q} \mathit{cost}\Big(\ell + 1, \rho_1^{\mathit{light}}[I_j], ..., \rho_{h_{\ell+1}-1}^{\mathit{light}}[I_j], \rho_{h_{\ell+1}}, \rho_{h_{\ell+1}+1}^{\mathit{light}}[I_j], ..., \rho_d^{\mathit{light}}[I_j]\Big) \qquad (11)
\end{aligned}$$

where q is the number of disjoint intervals that $\textsc{Join}(h_\ell, \rho_1, ..., \rho_d)$ uses to divide $\mathit{dom}(A_\ell)$ (see Section 3.2.2). By the balancing condition in Section 3.2.2, it must hold that

$$\begin{aligned}
q &= O(1 + |\rho_1|/\tau_{h_{\ell+1}}) \\
\text{(by (4))} &= O(1 + \tau_{h_\ell}/\tau_{h_{\ell+1}}) \\
&= O(\mu_\ell). \qquad (12)
\end{aligned}$$

The rest of the section is devoted to analyzing the above non-conventional recurrence.

3.4 Solving the Recurrence

Our objective is to prove that $\mathit{cost}(1, r_1, ..., r_d)$—which gives the number of I/Os of our LW enumeration algorithm—is as claimed in Theorem 2. We will do so by resorting to the recursion tree $\mathcal{T}$ (as is defined in Section 3.3). Specifically, for each node $\textsc{Join}(\ell, \rho_1, ..., \rho_d)$, we associate it with cost

- $O(d + \mathit{sort}(d\sum_{i=1}^{d} |\rho_i|))$, if it is a leaf in $\mathcal{T}$. We account for the two terms in different ways. First, the term $O(d)$ is charged as the *factual cost* on the leaf itself. On the other hand, on every tuple in ρ_i (of all $i \in [1, d]$), we charge a *nominal cost* of $O(d)$. In this way, the second term $O(\mathit{sort}(d\sum_{i=1}^{d} |\rho_i|))$ equals $O(\mathit{sort}(x))$, where x is the sum of the nominal costs of all the tuples in $\rho_1, ..., \rho_d$.
- $O(d \cdot \mu_\ell + \mathit{sort}(d^2 \cdot \mu_\ell \cdot |\rho_{h_{\ell+1}}| + d\sum_{i=1}^{d} |\rho_i|))$, if it is an internal node in $\mathcal{T}$. Again, we account for the terms differently. The term $O(d \cdot \mu_\ell)$ is charged as the factual cost of the node itself. On every tuple in ρ_i for $i \in [1, d]$, we charge a nominal cost of $O(d)$, whereas on every tuple

in $\rho_{h_{\ell+1}}$, we charge an additional nominal cost of $O(d^2 \cdot \mu_\ell)$. The term $sort(d^2 \cdot \mu_\ell \cdot |\rho_{h_{\ell+1}}| + d\sum_{i=1}^{d} |\rho_i|))$ equals $O(sort(x'))$, where x' is the sum of the nominal costs of all the tuples in $\rho_1, ..., \rho_d$ incurred this way.

The above strategy allows us to bound $cost(1, r_1, ..., r_d)$ by adding up two costs:

- The sum of the factual costs of all nodes in $\mathcal{T}$;
- $sort(X)$, where X is the sum of the total nominal costs charged on all the tuples in $r_1, r_2, ..., r_d$ across all levels of $\mathcal{T}$.[1]

Next, we will concentrate on each bullet in turn.

Bounding the Factual Costs. Let us now focus on Bullet 1. The key is to bound the number m_ℓ of nodes at each level $\ell \in [1, w]$ of $\mathcal{T}$. We say that a level-ℓ call $\text{JOIN}(h_\ell, \rho_1, ..., \rho_d)$ *underflows* if $|\rho_1| < \tau_{h_\ell}/2$; otherwise, it is *ordinary.* Consider all the calls $\text{JOIN}(h_\ell, \rho_1, ..., \rho_d)$ at level ℓ. The sets ρ_1 (i.e., the first parameter) of those calls are disjoint. Hence, there can be at most $O(n_1/\tau_{h_\ell})$ ordinary calls at level ℓ. Moreover, if $\ell < w$, then a level-ℓ call creates at most one underflowing call at level $\ell + 1$. This discussion indicates that, for each $\ell \in [2, w]$:

$$m_\ell = O\left(m_{\ell-1} + \frac{n_1}{\tau_{h_\ell}}\right) = O\left(\sum_{i=1}^{\ell} \frac{n_1}{\tau_{h_i}}\right) = O\left(\frac{n_1}{\tau_{h_\ell}}\right), \tag{13}$$

where the second equality used $m_1 = 1 = n_1/\tau_{h_1}$, and the last equality used the fact that $\tau_{h_i} > 2\tau_{h_{i+1}}$ for every $i \in [1, w-1]$.

Therefore, the sum of the factual costs of all the nodes in $\mathcal{T}$ is bounded by

$$\begin{aligned}
O\left(d \cdot m_w + \sum_{\ell=1}^{w-1} d \cdot \mu_\ell \cdot m_\ell\right) &= O\left(\frac{d \cdot n_1}{\tau_{h_w}} + \sum_{\ell=1}^{w-1} d \cdot \mu_\ell \cdot \frac{n_1}{\tau_{h_\ell}}\right) \\
\text{(by (8))} \quad &= O\left(\frac{d \cdot n_1}{\tau_{h_w}} + \sum_{\ell=1}^{w-1} d \cdot \frac{\tau_{\tau_{h_\ell}}}{\tau_{h_{\ell+1}}} \cdot \frac{n_1}{\tau_{h_\ell}}\right) \\
&= O\left(\frac{d \cdot n_1}{\tau_{h_w}} + \frac{d \cdot n_1}{\tau_{h_w}}\right) \\
\text{(by } \tau_{h_w} = M/d) \quad &= O\left(\frac{d^2 n_1}{M}\right).
\end{aligned} \tag{14}$$

Bounding X. Let us consider a single tuple t in an arbitrary input relation r_i (for any $i \in [1, d]$). We aim to bound the sum of all the nominal costs charged on t, across all the nodes of $\mathcal{T}$. Consider a level-ℓ call $(1 \le \ell \le w)$ $\text{JOIN}(h_\ell, \rho_1, ..., \rho_d)$ in $\mathcal{T}$. We say that t *participates in* the call if $t \in \rho_i$. Obviously, t has a nominal cost on this node if and only if t participates in it.

For $i \in [2, d]$, define a value $L_i \in [0, w]$ as follows:

- $L_i = \ell$ if i is the axis of the calls at level-ℓ of $\mathcal{T}$ for some $\ell \in [2, w]$, i.e., $h_\ell = i$;
- $L_i = 0$, otherwise.

We make the following observation on the participation of t in different levels:

[1]Notice that $sort(x) + sort(x') \le sort(x + x')$ for any $x, x' > 0$.

Lemma 6. *If $L_i = 0$, then t participates at most once at level ℓ for all $\ell \in [1, w]$. Otherwise, t participates*

- *at most once at level-ℓ for each $\ell \in [1, L_i - 1]$;*
- *$O(\mu_{L_i - 1})$ times at level ℓ for each $\ell \in [L_i, w]$.*

Proof. The lemma follows from how t is passed from a call to its descendants in $\mathcal{T}$. Let $\text{JOIN}(h_\ell, \rho_1, ..., \rho_d)$ be a level-ℓ call that t participates in. If $h_{\ell+1} \neq i$, then t participates in *at most one* of the call's child nodes in $\mathcal{T}$. Otherwise ($h_{\ell+1} = i$, and hence, $L_i = \ell + 1$ by definition), t may participate in *all* of the call's child nodes in $\mathcal{T}$. From (12), we know that $\text{JOIN}(h_\ell, \rho_1, ..., \rho_d)$ has $q = O(\mu_\ell) = O(\mu_{L_i - 1})$. □

Hence, the total nominal cost of t in the entire $\mathcal{T}$ equals:

- $O(d \cdot w) = O(d^2)$, if $L_i = 0$.
- $O(d \cdot L_i + d^2 \cdot \mu_{L_i-1} + d \cdot \mu_{L_i-1} \cdot (d - L_i + 1)) = O(d^2 \cdot \mu_{L_i-1})$, otherwise. Specifically, the term $O(d \cdot L_i)$ is due to the at most one participation of t at each level from 1 to $L_i - 1$, whereas the term $O(d \cdot \mu_{L_i-1} \cdot (d - L_i + 1))$ is due to the $O(\mu_{L_i-1})$ participations at each level from L_i to d. The term $O(d^2 \cdot \mu_{L_i-1})$ is due to the additional cost charged on t at level $L_i - 1$.

Therefore, X—the sum of the total nominal cost of *all* tuples—is bounded by

$$\begin{aligned} O\Big(\sum_{i\in[1,d] \text{ s.t. } L_i \neq 0} d^2 \mu_{L_i} n_i + \sum_{i=1}^{d} d^2 n_i\Big) &= O\Big(\sum_{\ell=2}^{w} d^2 \mu_{\ell-1} n_{h_\ell} + d^2 \sum_{i=1}^{d} n_i\Big) \\ \text{(by Lemma 5)} &= O\Big(\sum_{\ell=2}^{w} U d^{2+\frac{1}{d-1}} + d^2 \sum_{i=1}^{d} n_i\Big) \\ &= O\Big(d^{3+\frac{1}{d-1}} U + d^2 \sum_{i=1}^{d} n_i\Big). \end{aligned}$$

Summary. Combining the above equation with (2) and plugging in the definition of U in (14), we now complete the whole proof of Theorem 2.

3.5 A Lower Bound Remark for Constant d

When the number d of attributes is a constant, our LW enumeration algorithm guarantees I/O cost $O(sort((\prod_{i=1}^{d} n_i/M)^{\frac{1}{d-1}} + \sum_{i=1}^{d} n_i))$. On the other hand, using an argument similar to those in [8, 14], we can establish an I/O lower bound of $\Omega(n^{\frac{d}{d-1}}/(BM^{\frac{1}{d-1}}))$ in the scenario where $n_1 = n_2 = ... = n_d = n$, on the class of *witnessing algorithms*. Our algorithm, which is in this class, is thus asymptotically optimal up to a logarithmic factor. Next, we present a proof of the lower bound.

We will refer to a tuple in $r_1, r_2, ...,$ or r_d as an *input tuple*. A witnessing algorithm is modeled as follows. A disk block has the capacity to store precisely B input tuples. At the beginning, the memory is empty, while all the input tuples reside in the disk. At any moment, the algorithm is allowed to keep at most M input tuples in the memory. At each step, it is permitted three operations:

- *Read I/O*: fetch B input tuples from a disk block into the memory;

- *Write I/O*: write B input tuples in the memory to a disk block;
- Perform *emit*(t^*) for a result tuple $t^* = r_1 \bowtie ... \bowtie r_d$, provided that the d input tuples that produce t^* are all in the memory currently.

Let us consider a hard input to the LW enumeration problem where the size of $r_1 \bowtie ... \bowtie r_d$ equals $n^{d/(d-1)}$ (such an input indeed exists [4]). Suppose that an algorithm solves the problem on this input using H read I/Os (write I/Os are for free in our analysis). Chop the sequence of these read I/Os into *epochs* where each epoch is a subsequence of M/B I/Os, except possibly the last one. As shown later, during each epoch, *emit*(.) can only be called $O(M^{d/(d-1)})$ times. This implies that

$$\frac{H}{M/B} \cdot O(M^{d/(d-1)}) \quad \geq \quad n^{d/(d-1)}$$

which suggests $H = \Omega(n^{\frac{d}{d-1}}/(BM^{\frac{1}{d-1}}))$.

It remains to explain why there can be only $O(M^{d/(d-1)})$ tuple emissions during an epoch. Define S to be a set of input tuples defined as follows. S includes (i) all the input tuples already in the memory at the beginning of the epoch, and (ii) all the input tuples in the (at most) M/B blocks read in the epoch. Clearly $|S| \leq 2M$. Every tuple emitted during the epoch must be produced by the tuples in S. Suppose that S contains x_i ($i \in [1, d]$) tuples from r_i. According to [4], these input tuples can produced at most $(\prod_{i=1}^{d} x_i)^{1/(d-1)}$ tuples in $r_1 \bowtie ... \bowtie r_d$, which is at most $(2M/d)^{d/(d-1)} = O(M^{d/(d-1)})$ under the constraint $\sum_{i=1}^{d} x_i \leq 2M$.

4 A Faster Algorithm for Arity 3

The algorithm developed in the previous section solves the LW enumeration problem for any $d \leq M/2$. In this section, we focus on $d = 3$, and leverage intrinsic properties of this special instance to design a faster algorithm, which will establish Theorem 3 (and hence, also Corollaries 1 and 2). Specifically, the input consists of three relations: $r_1(A_2, A_3)$, $r_2(A_1, A_3)$, and $r_3(A_1, A_2)$; and the goal is to emit all the tuples in the result of $r_1 \bowtie r_2 \bowtie r_3$.

As before, for each $i \in [1, 3]$, set $n_i = |r_i|$, and denote by $dom(A_i)$ the domain of A_i. Without loss of generality, we assume that $n_1 \geq n_2 \geq n_3$.

4.1 Basic Algorithms

Let us start with:

Lemma 7. *If $r_1(A_2, A_3)$ and $r_2(A_1, A_3)$ have been sorted by A_3, the 3-arity LW enumeration problem can be solved in $O(1 + \frac{(n_1+n_2)n_3}{MB} + \frac{1}{B}\sum_{i=1}^{3} n_i)$ I/Os.*

Proof. If $n_3 \leq M$, we can achieve the purpose stated in the lemma using the small-join algorithm of Lemma 2 with straightforward modifications (e.g., apparently sorting is not required). When $n_3 > M$, we simply chop r_3 into subsets of size M, and then repeat the above small-join algorithm $\lceil n_3/M \rceil$ times. $\square$

We call $r_1 \bowtie r_2 \bowtie r_3$ an A_1-*point join* if both conditions below are fulfilled:

- all the A_1 values in $r_2(A_1, A_3)$ are the same;

- $r_1(A_2, A_3)$ and $r_2(A_1, A_3)$ are sorted by A_3.

Lemma 8. *Given an A_1-point join, we can emit all its result tuples in $O(1 + \frac{n_1 n_3}{MB} + \frac{1}{B}\sum_{i=1}^{3} n_i)$ I/Os.*

Proof. We first obtain $r'(A_1, A_2, A_3) = r_1 \bowtie r_2$, and store all the tuples of r' into the disk. Since all the tuples in r_2 have the same A_1-value, their A_3-values must be distinct. Hence, each tuple in r_1 can be joined with at most one tuple in r_2, implying that $|r'| \le n_1$. Utilizing the fact that r_1 and r_2 are both sorted on A_3, r' can be produced by a synchronous scan over r_1 and r_2 in $O(1 + (n_1 + n_2)/B)$ I/Os.

Then, we use the classic *blocked nested loop* (BNL) algorithm to perform the join $r' \bowtie r_3$ (which equals $r_1 \bowtie r_2 \bowtie r_3$). The only difference is that, whenever BNL wants to write a block of $O(B)$ result tuples to the disk, we skip the write but simply emit those tuples. The BNL performs $O(1 + \frac{|r'| n_3}{MB} + \frac{r' + n_3}{B})$ I/Os. The lemma thus follows. □

Symmetrically, we call $r_1 \bowtie r_2 \bowtie r_3$ an *A_2-point join* if

- all the A_2 values in $r_1(A_2, A_3)$ are the same.
- $r_1(A_2, A_3)$ and $r_2(A_1, A_3)$ are sorted by A_3.

Lemma 9. *Given an A_2-point join, we can emit all its result tuples in $O(1 + \frac{n_2 n_3}{MB} + \frac{1}{B}\sum_{i=1}^{3} n_i)$ I/Os.*

Proof. Symmetric to Lemma 8. □

4.2 3-Arity LW Enumeration Algorithm

Next, we give our general algorithm for LW enumeration with $d = 3$. We will focus on $n_1 \ge n_2 \ge n_3 \ge M$; if $n_3 < M$, the algorithm in Lemma 7 already solves the problem in linear I/Os after sorting.

Set:

$$\theta_1 = \sqrt{\frac{n_1 n_3 M}{n_2}}, \text{ and } \theta_2 = \sqrt{\frac{n_2 n_3 M}{n_1}}. \tag{15}$$

For values $a_1 \in dom(A_1)$ and $a_2 \in dom(A_2)$, define:

$$\begin{aligned} freq(a_1, r_3) &= |\{t \in r_3 \mid t[A_1] = a_1\}| \\ freq(a_2, r_3) &= |\{t \in r_3 \mid t[A_2] = a_2\}|. \end{aligned}$$

Now we introduce:

$$\begin{aligned} \Phi_1 &= \{a_1 \in dom(A_1) \mid freq(a_1, r_3) > \theta_1\} \\ \Phi_2 &= \{a_2 \in dom(A_2) \mid freq(a_2, r_3) > \theta_2\}. \end{aligned}$$

Let t^* be a result tuple of $r_1 \bowtie r_2 \bowtie r_3$. We can classify t^* into one of the following categories:

- *Heavy-heavy*: $t^*[A_1] \in \Phi_1$ and $t^*[A_2] \in \Phi_2$
- *Heavy-light*: $t^*[A_1] \in \Phi_1$ and $t^*[A_2] \notin \Phi_2$
- *Light-heavy*: $t^*[A_1] \notin \Phi_1$ and $t^*[A_2] \in \Phi_2$

- *Light-light*: $t^*[A_1] \notin \Phi_1$ and $t^*[A_2] \notin \Phi_2$.

We will emit each type of tuples separately, after a partitioning phase, as explained in the sequel.

Partitioning r_3. Define:

$$\begin{aligned}
r_3^{heavy,heavy} &= \{t \in r_3 \mid t[A_1] \in \Phi_1, t[A_2] \in \Phi_2\} \\
r_3^{heavy,light} &= \{t \in r_3 \mid t[A_1] \in \Phi_1, t[A_2] \notin \Phi_2\} \\
r_3^{light,heavy} &= \{t \in r_3 \mid t[A_1] \notin \Phi_1, t[A_2] \in \Phi_2\} \\
r_3^{light,light} &= \{t \in r_3 \mid t[A_1] \notin \Phi_1, t[A_2] \notin \Phi_2\} \\
r_3^{light,-} &= r_3^{light,heavy} \cup r_3^{light,light} \\
r_3^{-,light} &= r_3^{heavy,light} \cup r_3^{light,light}.
\end{aligned}$$

Divide $dom(A_1)$ into $q_1 = O(1 + n_3/\theta_1)$ disjoint intervals I_1^1, I_2^1, ..., $I_{q_1}^1$ with the following properties: (i) $I_1^1, I_2^1, ..., I_{q_1}^1$ are in ascending order, and (ii) for each $j \in [1, q_1]$, $r_3^{light,-}$ has at most $2\theta_1$ tuples whose A_1-values fall in I_j^1. Similarly, we divide $dom(A_2)$ into $q_2 = O(1 + n_3/\theta_2)$ disjoint intervals $I_1^2, I_2^2, ..., I_{q_2}^2$ with the following properties: (i) $I_1^2, I_2^2, ..., I_{q_2}^2$ are in ascending order, and (ii) for each $j \in [1, q_2]$, $r_3^{-,light}$ has at most $2\theta_2$ tuples whose A_2-values fall in I_j^2.

We now define several partitions of r_3:

- For each $a_1 \in \Phi_1$ and $a_2 \in \Phi_2$, let $r_3^{heavy,heavy}[a_1, a_2]$ be the (*only*) tuple t in $r_3^{heavy,heavy}$ with $t[A_1] = a_1$ and $t[A_2] = a_2$.

- For each $a_1 \in \Phi_1$ and $j \in [1, q_2]$, let $r_3^{heavy,light}[a_1, I_j^2]$ be the set of tuples t in $r_3^{heavy,light}$ with $t[A_1] = a_1$ and $t[A_2]$ in I_j^2.

- For each $j \in [1, q_1]$ and $a_2 \in \Phi_2$, let $r_3^{light,heavy}[I_j^1, a_2]$ be the set of tuples t in $r_3^{light,heavy}$ with $t[A_1]$ in I_j^1 and $t[A_2] = a_2$.

- For each $j_1 \in [1, q_1]$ and $j_2 \in [1, q_2]$, let $r_3^{light,light}[I_{j_1}^1, I_{j_2}^2]$ be the set of tuples t in $r_3^{light,light}$ with $t[A_1]$ in I_j^1 and $t[A_2]$ in I_j^2.

It is rudimentary to produce all the above partitions with $O(sort(n_3))$ I/Os in total.

Partitioning r_1 and r_2. Let:

$$\begin{aligned}
r_1^{heavy} &= \text{set of tuples } t \text{ in } r_1 \text{ s.t. } t[A_2] \in \Phi_2 \\
r_1^{light} &= \text{set of tuples } t \text{ in } r_1 \text{ s.t. } t[A_2] \notin \Phi_2 \\
r_2^{heavy} &= \text{set of tuples } t \text{ in } r_2 \text{ s.t. } t[A_1] \in \Phi_1 \\
r_2^{light} &= \text{set of tuples } t \text{ in } r_2 \text{ s.t. } t[A_1] \notin \Phi_1
\end{aligned}$$

We now define several partitions of r_1:

- For each $a_2 \in \Phi_2$, let $r_1^{heavy}[a_2]$ be the set of tuples t in r_1^{heavy} with $t[A_2] = a_2$.

- For each $j \in [1, q_2]$, let $r_1^{light}[I_j^2]$ be the set of tuples t in r_1^{light} with $t[A_2]$ in I_j^2.

Similarly, we define several partitions of r_2:

- For each $a_1 \in \Phi_1$, let $r_2^{heavy}[a_1]$ be the set of tuples t in r_2^{heavy} with $t[A_1] = a_1$.
- For each $j \in [1, q_1]$, let $r_2^{light}[I_j^1]$ be the set of tuples t in r_2^{light} with $t[A_1]$ in I_j^1.

It is rudimentary to produce the above partitions using $O(sort(n_1 + n_2 + n_3))$ I/Os in total. With the same cost, we make sure that all these partitions are sorted by A_3.

Enumerating Result Tuples. We emit each type of tuples as follows:

- *Heavy-heavy*: For each $a_1 \in \Phi_1$ and each $a_2 \in \Phi_2$, apply Lemma 7 to emit the result of $r_1^{heavy}[a_2] \bowtie r_2^{heavy}[a_1] \bowtie r_3^{heavy,heavy}[a_1, a_2]$.
- *Heavy-light*: For each $a_1 \in \Phi_1$ and each $j \in [1, q_2]$, apply Lemma 8 to emit the result of the A_1-point join $r_1^{light}[I_j^2] \bowtie r_2^{heavy}[a_1] \bowtie r_3^{heavy,light}[a_1, I_j^2]$.
- *Light-heavy*: For each $j \in [1, q_1]$ and each $a_2 \in \Phi_2$, apply Lemma 9 to emit the result of the A_2-point join $r_1^{heavy}[a_2] \bowtie r_2^{light}[I_j^1] \bowtie r_3^{light,heavy}[I_j^1, a_2]$.
- *Light-light*: For each $j_1 \in [1, q_1]$ and each $j_2 \in [1, q_2]$, apply Lemma 7 to emit the result of $r_1^{light}[I_{j_2}^2] \bowtie r_2^{light}[I_{j_1}^1] \bowtie r_3^{light,light}[I_{j_1}^1, I_{j_2}^2]$.

4.3 Analysis

We now analyze the algorithm of Section 4.2, assuming $n_1 \geq n_2 \geq n_3 \geq M$. First, it should be clear that

$$\begin{aligned}
|\Phi_1| &\leq \frac{n_3}{\theta_1} = \sqrt{\frac{n_2 n_3}{n_1 M}} \\
|\Phi_2| &\leq \frac{n_3}{\theta_2} = \sqrt{\frac{n_1 n_3}{n_2 M}} \\
q_1 &= O\Big(1 + \frac{n_3}{\theta_1}\Big) = O\Big(1 + \sqrt{\frac{n_2 n_3}{n_1 M}}\Big) \\
q_2 &= O\Big(1 + \frac{n_3}{\theta_2}\Big) = O\Big(\sqrt{\frac{n_1 n_3}{n_2 M}}\Big).
\end{aligned}$$

By Lemma 7, the cost of red-red emission is bounded by (remember that $r_3^{heavy,heavy}[a_1, a_2]$ has only 1 tuple):

$$\begin{aligned}
&\sum_{a_1,a_2} O\Big(1 + \frac{|r_1^{heavy}[a_2]| + |r_2^{heavy}[a_1]|}{B}\Big). \\
=\ & O\Big(|\Phi_1||\Phi_2| + \sum_{a_2} \frac{|r_1^{heavy}[a_2]||\Phi_1|}{B} + \sum_{a_1} \frac{|r_2^{heavy}[a_1]||\Phi_2|}{B}\Big) \\
=\ & O\Big(\frac{n_3}{M} + \frac{n_1|\Phi_1|}{B} + \frac{n_2|\Phi_2|}{B}\Big) = O\Big(\frac{\sqrt{n_1 n_2 n_3}}{B\sqrt{M}}\Big).
\end{aligned}$$

By Lemma 8, the cost of red-blue emission is bounded by:

$$\begin{aligned}
&\sum_{a_1,j} O\Big(1 + \frac{\big|r_1^{light}[I_j^2]\big|\big|r_3^{heavy,light}[a_1, I_j^2]\big|}{MB} + \frac{\big|r_1^{light}[I_j^2]\big| + \big|r_2^{heavy}[a_1]\big| + \big|r_3^{heavy,light}[a_1, I_j^2]\big|}{B}\Big). \\
&= O\Big(|\Phi_1|q_2 + \sum_j \frac{\big|r_1^{light}[I_j^2]\big| \sum_{a_1} \big|r_3^{heavy,light}[a_1, I_j^2]\big|}{MB} \\
&+ \frac{|\Phi_1| \sum_j \big|r_1^{light}[I_j^2]\big|}{B} + \frac{q_2 \sum_{a_1} \big|r_2^{heavy}[a_1]\big|}{B} + \frac{n_3}{B}\Big). \qquad (16)
\end{aligned}$$

Observe that $\sum_{a_1} \big|r_3^{heavy,light}[a_1, I_j^2]\big|$ is the total number of tuples in $r_3^{heavy,light}$ whose A_2-values fall in I_j^2. By the way $I_1^2, ..., I_{q_2}^2$ are constructed, we know:

$$\sum_{a_1} \big|r_3^{heavy,light}[a_1, I_j^2]\big| \le 2\theta_2.$$

(16) is thus bounded by:

$$\begin{aligned}
&O\Big(\frac{n_3}{M} + \sum_j \frac{\big|r_1^{light}[I_j^2]\big|\theta_2}{MB} + \frac{|\Phi_1|n_1}{B} + \frac{q_2 n_2}{B} + \frac{n_3}{B}\Big) \\
= \quad &O\Big(\frac{n_1\theta_2}{MB} + \frac{|\Phi_1|n_1}{B} + \frac{q_2 n_2}{B} + \frac{n_3}{B}\Big) = O\Big(\frac{\sqrt{n_1 n_2 n_3}}{B\sqrt{M}}\Big).
\end{aligned}$$

A similar argument shows that the cost of blue-red emission is bounded by $O(\frac{\sqrt{n_1 n_2 n_3}}{B\sqrt{M}} + \frac{n_1}{B})$. Finally, by Lemma 7, the cost of blue-blue emission is bounded by:

$$\begin{aligned}
\sum_{j_1,j_2} \quad &O\Big(1 + \frac{\big(\big|r_1^{light}[I_{j_2}^2]\big| + \big|r_2^{light}[I_{j_1}^1]\big|\big)\big|r_3^{light,light}[I_{j_1}^1, I_{j_2}^2]\big|}{MB} \\
&+ \frac{\big|r_1^{light}[I_{j_2}^2]\big| + \big|r_2^{light}[I_{j_1}^1]\big| + \big|r_3^{light,light}[I_{j_1}^1, I_{j_2}^2]\big|}{B}\Big). \qquad (17)
\end{aligned}$$

Let us analyze each term of (17) in turn. First:

$$\sum_{j_1,j_2} \big|r_1^{light}[I_{j_2}^2]\big|\big|r_3^{light,light}[I_{j_1}^1, I_{j_2}^2]\big| \quad = \quad \sum_{j_2} \big|r_1^{light}[I_{j_2}^2]\big| \sum_{j_1} \big|r_3^{light,light}[I_{j_1}^1, I_{j_2}^2]\big| \qquad (18)$$

$\sum_{j_1} \big|r_3^{light,light}[I_{j_1}^1, I_{j_2}^2]\big|$ gives the number of tuples in $r_3^{light,light}$ whose A_2-values fall in I_j^2. By the way $I_1^2, ..., I_{q_2}^2$ are constructed, we know:

$$\sum_{j_1} \big|r_3^{light,light}[I_{j_1}^1, I_{j_2}^2]\big| \le 2\theta_2.$$

Therefore:

$$(18) = O\Big(\theta_2 \sum_{j_2} \big|r_1^{light}[I_{j_2}^2]\big|\Big) = O(n_1\theta_2).$$

Symmetrically, we have:

$$\sum_{j_1,j_2} \big|r_2^{light}[I_{j_1}^1]\big|\big|r_3^{light,light}[I_{j_1}^1, I_{j_2}^2]\big| = O(n_2\theta_1).$$

Thus, (17) is bounded by:

$$\begin{aligned}
& O\Big(q_1q_2 + \frac{n_1\theta_2 + n_2\theta_1}{MB} + \frac{q_1\sum_{j_2}\left|r_1^{light}[I^2_{j_2}]\right|}{B} + \frac{q_2\sum_{j_1}\left|r_2^{light}[I^1_{j_1}]\right|}{B} + \frac{n_3}{B}\Big) \\
= \; & O\Big(q_1q_2 + \frac{n_1\theta_2 + n_2\theta_1}{MB} + \frac{q_1n_1}{B} + \frac{q_2n_2}{B} + \frac{n_3}{B}\Big) \\
= \; & O\Big(\frac{\sqrt{n_1n_2n_3}}{B\sqrt{M}} + \frac{n_1}{B}\Big).
\end{aligned}$$

As already mentioned in Section 4.2, the partitioning phase requires $O(sort(\sum_{i=1}^{3} n_i))$ I/Os. We now complete the proof of Theorem 3.

5 Conclusions

Checking whether a relation r can be decomposed—as far as natural join is concerned—is extremely important in database systems, and is the key to normalization. This paper presents the first systematic study on I/O-efficient algorithms on this topic. Our results are three-fold. First, we proved that it is NP-hard to check whether r satisfies a *specific* join dependency J, even if all the relation schemas in J have only 2 attributes. Second, we presented an I/O-efficient algorithm for determining whether r has redundancy *at all*—namely, if there is *any* non-trivial J satisfied by r. Our algorithm in fact solves a type of joins known as Loomis-Whitney (LW) Joins. Third, by observing that the classic triangle enumeration problem is a special instance of LW-joins, we further enhanced our LW-join algorithm for this instance, and solved triangle enumeration with the optimal I/O cost under all problem parameters.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley Publishing Company, 1995.

[2] Alok Aggarwal and Jeffrey Scott Vitter. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM (CACM)*, 31(9):1116–1127, 1988.

[3] Lars Arge, Paolo Ferragina, Roberto Grossi, and Jeffrey Scott Vitter. On Sorting Strings in External Memory (Extended Abstract). In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 540–548, 1997.

[4] Albert Atserias, Martin Grohe, and Dániel Marx. Size Bounds and Query Plans for Relational Joins. *SIAM Journal of Computing*, 42(4):1737–1767, 2013.

[5] C Beeri and M Vardi. On the Complexity of Testing Implications of Data Dependencies. *Computer Science Report, Hebrew Univ*, 1980.

[6] Patrick C. Fischer and Don-Min Tsou. Whether a Set of Multivalued Dependencies Implies a Join Dependency is NP-Hard. *SIAM Journal of Computing*, 12(2):259–266, 1983.

[7] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[8] Xiaocheng Hu, Yufei Tao, and Chin-Wan Chung. I/O-Efficient Algorithms on Triangle Listing and Counting. *ACM Transactions on Database Systems (TODS)*, 39(4):27:1–27:30, 2014.

[9] Paris C. Kanellakis. On the Computational Complexity of Cardinality Constraints in Relational Databases. *Information Processing Letters (IPL)*, 11(2):98–101, 1980.

[10] David Maier. *The Theory of Relational Databases.* Computer Science Press, 1983.

[11] David Maier, Yehoshua Sagiv, and Mihalis Yannakakis. On the Complexity of Testing Implications of Functional and Join Dependencies. *Journal of the ACM (JACM)*, 28(4):680–695, 1981.

[12] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-Case Optimal Join Algorithms: [Extended Abstract]. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 37–48, 2012.

[13] Jean-Marie Nicolas. Mutual Dependencies and Some Results on Undecomposable Relations. In *Proceedings of Very Large Data Bases (VLDB)*, pages 360–367, 1978.

[14] Rasmus Pagh and Francesco Silvestri. The Input/Output Complexity of Triangle Enumeration. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 224–233, 2014.

[15] Todd L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *Proceedings of International Conference on Database Theory (ICDT)*, pages 96–106, 2014.