Join Dependency Testing, Loomis-Whitney Join, and Triangle Enumeration

Xiaocheng Hu Miao Qiao Yufei Tao

CUHK
Hong Kong

ABSTRACT

In this paper, we revisit two fundamental problems in database theory. The first one is called *join dependency (JD) testing*, where we are given a relation r and a JD, and need to determine whether the JD holds on r. The second problem is called *JD existence testing*, where we need to determine if there exists *any* non-trivial JD that holds on r.

We prove that JD testing is NP-hard even if the JD is defined *only* on binary relations (i.e., each with only two attributes). Unless P = NP, this result puts a negative answer to the question whether it is possible to efficiently test JDs defined exclusively on *small* (in terms of attribute number) relations. The question has been open since the classic NP-hard proof of Maier, Sagiv, and Yannakakis in JACM'81 which requires the JD to involve a relation of $\Omega(d)$ attributes, where d is the number of attributes in r.

For JD existence testing, the challenge is to minimize the computation cost because the problem is known to be solvable in polynomial time. We present a new algorithm for solving the problem I/O-efficiently in the external memory model. Our algorithm in fact settles the closely related *Loomis-Whitney (LW) enumeration problem*, and as a side product, achieves the optimal I/O complexity for the *triangle enumeration problem*, improving a recent result of Pagh and Silvestri in PODS'14.

Categories and Subject Descriptors

F.2.2 [**Analysis of algorithms and problem complexity**]: Nonnumerical Algorithms and Problems; H.2.4 [**Database Management**]: Systems—*Relational databases*

Keywords

Join Dependency; Loomis-Whitney Join; Triangle Enumeration

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 paper:

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

PODS'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

Http://dx.doi.org/10.1145/2745754.2745768.

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*. 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 ($1 \leq i \leq d$) 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 $d \leq M/2$, 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 $emit(\Delta)$ 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 $r_1 = r_2 = r_3 = E$ (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, that 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 $n_i \leq n = |r|$ ($1 \leq i \leq d$), 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.

Problems 2 and 3 become much more challenging in external memory (EM). The algorithm of [12] (similarly, also the algorithm of [4]) is unaware of data blocking, relies heavily on hashing, and 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

Section 2 will establish our first main result:

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 $n_i = |r_i|$. In Section 3, 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 $sort(x)$ *equals* $(x/B) \lg_{M/B}(x/B)$.

The main difficulty in obtaining the above theorem is that we *cannot* materialize the join result, because (as mentioned before) the result may have up to $(\Pi_{i=1}^{d} n_i)^{1/(d-1)}$ tuples such that writing them all to the disk may necessitate $\Omega(\frac{d}{B}(\Pi_{i=1}^{d} n_i)^{1/(d-1)})$ I/Os. This is why the problem is more challenging in EM (than in RAM where it is affordable, in fact even compulsory, to list out the entire join result [4, 12]). We overcome the challenge with a delicate piece of recursive machinery, and prove its efficiency through a non-trivial analysis.

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[1] with a vertex set V and an edge set E. Set $n = |V|$ and $m = |E|$. Without loss of generality, assume that each vertex $v \in V$ is uniquely identified by an integer id in $[1, n]$, denoted as $id(v)$. 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 $r_{i,j}$: $(id(u), id(v))$ and $(id(v), id(u))$. 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 $1 \le x, y \le n$.

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

Define:

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

For example, for $n = 3$, CLIQUE $= r_{1,2} \bowtie r_{1,3} \bowtie r_{2,3}$. In general, CLIQUE is a relation with schema R.

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

PROOF. *Direction If.* Assuming that CLIQUE is not empty, next we show that G has a Hamiltonian path. Let $(id(v_1), id(v_2), ..., id(v_n))$ be an arbitrary tuple in CLIQUE. It follows that:

[1] Recall that a graph is *simple* if it has at most one edge between any two vertices.

- For every $i \in [1, n-1]$, $(id(v_i), id(v_{i+1}))$ is a tuple in $r_{i,i+1}$, indicating that E has an edge between v_i and v_{i+1}.
- For every i, j such that $j \geq i+2$, $(id(v_i), id(v_j))$ is a tuple in $r_{i,j}$, indicating that $id(v_i) \neq id(v_j)$, i.e., $v_i \neq v_j$.

We thus have found a Hamiltonian path $v_1, v_2, ..., v_n$ in G.

Direction Only-If. Assuming that G has a Hamiltonian path, next we show that CLIQUE is not empty. Let $v_1, v_2, ..., v_n$ be any Hamiltonian path in G. It is easy to verify that $(id(v_1)$, $id(v_2)$, ..., $id(v_n))$ must appear in CLIQUE. □

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

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

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. The construction of r^* takes time polynomial to n (and hence, also to m because $m \leq n^2$).

Initially, r^* is empty. For *every* tuple t in *every* relation $r_{i,j}$ ($1 \leq i < j \leq n$), 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 (as mentioned before) 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 2. 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:

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

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

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

Direction If. Assuming that r^* satisfies J, next we 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.

Direction Only-If. Assuming that CLIQUE is empty, next we 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 2), 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 ($1 \leq i \leq d$) 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. The first situation 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.*

LEMMA 3. *Given a small join, we can emit all its result tuples in* $O(d + sort(d \sum_{i=1}^{d} n_i))$ *I/Os.*

PROOF. See appendix. □

The second scenario 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*.

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. See appendix. □

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. We will focus on $n_1 > 2M/d$; if $n_1 \le 2M/d$, simply apply Lemma 3 because this is a small-join scenario.

Define:

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

$$\tau_i = \frac{n_1 n_2 ... n_i}{(U \cdot d^{\frac{1}{d-1}})^{i-1}} \quad \text{for each } i \in [1, d]. \tag{2}$$

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

Our general algorithm is a recursive procedure 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:
$$|\rho_1| \le \tau_h. \tag{3}$$

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

3.2.1 Case $\tau_h \le 2M/d$

In this case, by the requirements of JOIN$(h, \rho_1, ..., \rho_d)$, it holds that $|\rho_1| \le \tau_h = O(M/d)$. Hence, we can directly apply the small-join algorithm in Lemma 3 to carry out the LW enumeration.

3.2.2 Case $\tau_h > 2M/d$

Denote by H the smallest integer in $[h+1, d]$ such that $\tau_H < \tau_h/2$. H always exists because $\tau_d = M/d < \tau_h/2$. Given a value $a \in dom(A_H)$, we define

$$freq(a) = \text{number of tuples } t \text{ in } \rho_1 \text{ with } t[A_H] = a.$$

Now we introduce:

$$\Phi = \{a \in dom(A_H) \mid freq(a) > \tau_H/2\}. \tag{4}$$

Let t^* be a result tuple of $\rho_1 \bowtie ... \bowtie \rho_d$. Conceptually, t^* is given a color: (i) *red*, if $t^*[A_H] \in \Phi$, or (ii) *blue*, otherwise.

Our strategy is to emit red and blue tuples separately. Towards this purpose, for each $i \in [1, d] \setminus \{H\}$, we partition ρ_i into:

$$\rho_i^{red} = \{\text{tuple } t \text{ in } \rho_i \mid t[A_H] \in \Phi\}$$
$$\rho_i^{blue} = \{\text{tuple } t \text{ in } \rho_i \mid t[A_H] \notin \Phi\}$$

To emit red tuples, it suffices to consider ρ_1^{red}, ..., ρ_{H-1}^{red}, ρ_H, ρ_{H+1}^{red}, ..., ρ_d^{red}. Likewise, to emit blue tuples, it suffices to consider ρ_1^{blue}, ..., ρ_{H-1}^{blue}, ρ_H, ρ_{H+1}^{blue}, ..., ρ_d^{blue}. Next, we will elaborate on how to do so.

Remark. The set Φ, as well as ρ_i^{red} and ρ_i^{blue} for each $i \in [1, d] \setminus \{H\}$, can be produced by sorting each ρ_i on A_H. More specifically, each element to be sorted is a tuple of $d-1$ values where d can be as large as $M/2$. 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.

Emitting Red Tuples. For every $a \in \Phi$, we aim to emit the red tuples t^* with $t^*[A_H] = a$ separately. Define for each $i \in [1, d] \setminus \{H\}$:

$$\rho_i^{red}[a] = \text{set of tuples } t \text{ in } \rho_i^{red} \text{ with } t[A_H] = a.$$

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

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

Emitting Blue Tuples. First, divide $dom(A_H)$ into $q = O(1 + |\rho_1|/\tau_H)$ *disjoint* intervals $I_1, I_2, ..., I_q$ with the following properties:

- $I_1, I_2, ..., I_q$ are in ascending order[2].
- For each $j \in [1, q]$, define:
$$\rho_1^{blue}[I_j] = \text{set of tuples in } \rho_1^{blue} \text{ whose } A_H\text{-values fall in } I_j$$
If $j < q$, we require $\tau_H/2 \le |\rho_1^{blue}[I_j]| \le \tau_H$. Regarding $\rho_1^{blue}[I_q]$, we require $1 \le |\rho_1^{blue}[I_q]| \le \tau_H$.

Because ρ_1 has been sorted by A_H, all the $I_1, ..., I_q$ and $\rho_1^{blue}[I_1], ..., \rho_1^{blue}[I_q]$ can all be obtained with one scan of ρ_1.

Next, for each $i \in [2, d] \setminus \{H\}$, we produce for each $j \in [1, q]$:

$$\rho_i^{blue}[I_j] = \text{set of tuples in } \rho_i^{blue} \text{ whose } A_H\text{-values fall in } I_j.$$

Because ρ_i^{blue} has been sorted by A_H, all the $\rho_i^{blue}[I_1]$, $\rho_i^{blue}[I_2]$, ..., $\rho_i^{blue}[I_q]$ can be obtained by scanning synchronously ρ_i^{blue} and $\{I_1, ..., I_q\}$ once.

Finally, to emit all the blue tuples, we simply recursively call our algorithm for each $j \in [1, q]$:

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

Note that the requirements for calling JOIN are fulfilled—in particular, $|\rho_1^{blue}[I_j]| \le \tau_H$, due to the way $I_1, ..., I_q$ were determined.

3.3 Analysis

Define a sequence of integers as follows:

- $h_1 = 1$;
- After h_i has been defined ($i \ge 1$):
  - if $\tau_{h_i} > 2M/d$, then define h_{i+1} as the smallest integer in $[1 + h_i, d]$ satisfying $\tau_{h_{i+1}} < \tau_{h_i}/2$;
  - otherwise, h_{i+1} is undefined.

Denote by w the largest integer with h_w defined.

Recall that our LW enumeration algorithm starts by calling the JOIN procedure with JOIN$(1, r_1, ..., r_d)$, which recursively makes

[2]An interval $[x, y]$ *precedes* another $[x', y']$ if $y < x'$.

$$f(\ell, \rho_1, ..., \rho_d) = \begin{cases} O(d) & \text{if } \ell = w \\ O(d \cdot \mu_\ell) + \sum_{j=1}^{q} f(\ell+1, \rho_1^{blue}[I_j], ..., \rho_{h_{\ell+1}-1}^{blue}[I_j], \rho_{h_{\ell+1}}, \rho_{h_{\ell+1}+1}^{blue}[I_j], ..., \rho_d^{blue}[I_j]) & \text{if } \ell < w \end{cases}$$

$$g(\ell, \rho_1, ..., \rho_d) = \begin{cases} d \sum_{i=1}^{d} |\rho_i| & \text{if } \ell = w \\ d^2 \mu_\ell |\rho_{h_{\ell+1}}| + d \sum_{i=1}^{d} |\rho_i| + \sum_{j=1}^{q} g(\ell+1, \rho_1^{blue}[I_j], ..., \rho_{h_{\ell+1}-1}^{blue}[I_j], \rho_{h_{\ell+1}}, \rho_{h_{\ell+1}+1}^{blue}[I_j], ..., \rho_d^{blue}[I_j]) & \text{if } \ell < w \end{cases}$$

Figure 1: Definitions of $f(h, \rho_1, ..., \rho_d)$ and $g(h, \rho_1, ..., \rho_d)$

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 3). 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$(h_\ell, \rho_1, ..., \rho_d)$. Our goal is to prove that $cost(1, r_1, ..., r_d)$ is as claimed in Theorem 2.

Case $\ell = w$. Lemma 3 immediately shows:

$$cost(w, \rho_1, ..., \rho_d) = O\Big(d + sort\Big(d \sum_{i=1}^{d} |\rho_i|\Big)\Big). \quad (5)$$

Case $\ell < w$. Define for $\ell \in [1, w-1]$:

$$\mu_\ell = 2\tau_{h_\ell}/\tau_{h_{\ell+1}}.$$

Consider the set Φ defined in (4). Recall that for every $a \in \Phi$, *freq*$(a) > \tau_{h_{\ell+1}}/2$. Hence:

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

where the second inequality is due to (3).

For emitting red tuples, the cost is dominated by that of the point-join algorithm whose total I/O cost, by Lemma 4, is bounded by:

$$O\Big(\sum_{a \in \Phi} \Big(d + sort\Big(d^2 |\rho_{h_{\ell+1}}| + d \sum_{i \in [1,d] \setminus \{h_{\ell+1}\}} \Big|\rho_i^{red}[a]\Big|\Big)\Big)\Big)$$
$$= O\Big(d|\Phi| + sort\Big(d^2 |\Phi||\rho_{h_{\ell+1}}| + d \sum_{i=1}^{d} |\rho_i|\Big)\Big)$$
$$= O\Big(d \cdot \mu_\ell + sort\Big(d^2 \mu_\ell |\rho_{h_{\ell+1}}| + d \sum_{i=1}^{d} |\rho_i|\Big)\Big). \quad (6)$$

The cost of emitting blue tuples comes from recursion. Therefore, we can establish a recurrence:

$$cost(\ell, \rho_1, ..., \rho_d)$$
$$= (6) + \sum_{j=1}^{q} cost\Big(\ell+1, \rho_1^{blue}[I_j], ..., \rho_{h_{\ell+1}-1}^{blue}[I_j],$$
$$\rho_{h_{\ell+1}}, \rho_{h_{\ell+1}+1}^{blue}[I_j], ..., \rho_d^{blue}[I_j]\Big). \quad (7)$$

Recall that q is the number of disjoint intervals that JOIN$(h_\ell, \rho_1, ..., \rho_d)$ uses to divide $dom(A_\ell)$ for blue tuple emission (see Section 3.2).

The rest of the subsection is devoted to solving this non-conventional recurrence. Let functions $f(\ell, \rho_1, ..., \rho_d)$ and $g(\ell, \rho_1, ..., \rho_d)$ be as defined in Figure 1. The following proposition is fundamental:

PROPOSITION 1. $cost(\ell, \rho_1, ..., \rho_d) = f(\ell, \rho_1, ..., \rho_d) + O(sort(g(\ell, \rho_1, ..., \rho_d)))$.

PROOF. By the convexity of function $sort(x)$. □

To prove Theorem 2, our target is to give an upper bound on $cost(1, r_1, ..., r_d) = f(1, r_1, ..., r_d) + O(sort(g(1, r_1, ..., r_d)))$.

3.3.1 Bounding $f(1, r_1, ..., r_d)$

Define m_ℓ as the total number of level-ℓ calls in $\mathcal{T}$. Each level-ℓ call contributes $O(d \cdot \mu_\ell)$ I/Os to $f(1, r_1, ..., r_d)$ (see Figure 1).[3] Hence:

$$f(1, r_1, ..., r_d) = \sum_{\ell=1}^{w} O(m_\ell \cdot d \cdot \mu_\ell). \quad (8)$$

We say that a level-ℓ call JOIN$(h_\ell, \rho_1, ..., \rho_d)$ *underflows* if $|\rho_1| < \tau_{h_\ell}/2$; otherwise, we say that it is *ordinary*. Consider all the calls JOIN$(h_\ell, \rho_1, ..., \rho_d)$ at level ℓ. The sets ρ_1 in the first parameters 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 $\ell + 1$. These facts indicate that, for each $\ell \in [2, w]$:

$$m_\ell = O\left(m_{\ell-1} + \frac{n_1}{\tau_{h_\ell}}\right) = O\left(\sum_{i=1}^{\ell} \frac{n_1}{\tau_{h_i}}\right) = O\left(\frac{n_1}{\tau_{h_\ell}}\right), \quad (9)$$

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 $i \in [1, w-1]$.

Applying $\tau_{h_w} = M/d$, we get from (9):

$$m_w = O(dn_1/M).$$

Moreover, for each $\ell \in [1, w-1]$:

$$m_\ell \mu_\ell = O\left(\frac{n_1}{\tau_{h_\ell}}\right) \frac{2\tau_{h_\ell}}{\tau_{h_{\ell+1}}} = O\left(\frac{n_1}{\tau_{h_{\ell+1}}}\right).$$

[3]Here define a boundary dummy $\mu_w = 1$.

We can now derive from (8):

$$\begin{aligned} f(1, r_1, ..., r_d) &= O\left(\frac{d^2 n_1}{M} + \sum_{\ell=1}^{w-1} \frac{d \cdot n_1}{\tau_{h_{\ell+1}}}\right) \\ &= O\left(\frac{d^2 n_1}{M} + \frac{d n_1}{\tau_{h_w}}\right) \\ &= O\left(\frac{d^2 n_1}{M}\right). \end{aligned} \quad (10)$$

3.3.2 Bounding $g(1, r_1, ..., r_d)$

Figure 1 shows that, in $\mathcal{T}$, each level-ℓ ($\ell < w$) call JOIN$(h_\ell, \rho_1, ..., \rho_d)$ contributes $d^2 \mu_\ell |\rho_{h_{\ell+1}}| + d \sum_{i=1}^{d} |\rho_i|$ to $g(1, r_1, ..., r_d)$. We can amortize the contribution onto the tuples in $\rho_1, ..., \rho_d$, such that:

- Each tuple in $\rho_{h_{\ell+1}}$ contributes $d^2 \mu_\ell$ to $g(1, r_1, ..., r_d)$;
- Each tuple in any other relation ρ_i ($i \neq h_{\ell+1}$) contributes d to $g(1, r_1, ..., r_d)$.

Similarly, for every level-w call JOIN$(h_w, \rho_1, ..., \rho_d)$, each tuple in ρ_1, ..., ρ_d contributes d to $g(1, r_1, ..., r_d)$.

Our strategy for bounding $g(1, r_1, ..., r_d)$ is to sum up the largest possible contribution made by *each* individual tuple in the input relations $r_1, ..., r_d$. For this purpose, given a value $i \in [1, d]$, we define L_i as follows:

- $L_1 = 0$;
- If $i \geq 2$ but no call in the entire $\mathcal{T}$ has axis i, then $L_i = 0$;
- Otherwise, suppose that the level-ℓ calls of T have axis $h_\ell = i$; then we define $L_i = \ell - 1$.

Now, let us concentrate on a single tuple t in an arbitrary input relation r_i (for any $i \in [1, d]$). Consider a level-ℓ call ($1 \leq \ell \leq w$) JOIN$(h_\ell, \rho_1, ..., \rho_d)$ in $\mathcal{T}$. We say that t *participates in* the call if $t \in \rho_i$. If t does not participate in the call, then t contributes nothing to $g(1, r_1, ..., r_d)$. Otherwise, the contribution of t depends on whether $h_{\ell+1}$ happens to be i. As explained earlier, if $h_{\ell+1} = i$, t contributes $d^2 \mu_\ell$, or else t contributes d.

Denote by $\gamma_\ell(t)$ the number of level-ℓ calls that t participates in; specially, define $\gamma_0(t) = 0$. Then, the sequence $L_1, L_2, ..., L_d$ defined earlier allows us to represent concisely the total contribution of t as

$$\gamma_{L_i}(t) \cdot d^2 \mu_{L_i} + \sum_{\ell \in [1,w] \setminus L_i} \gamma_\ell(t) \cdot d \quad (11)$$

defining a boundary dummy value $\mu_0 = 1$.

LEMMA 5. *If $L_i = 0$, then $\gamma_\ell(t) \leq 1$ for all $\ell \in [1, w]$. If $L_i \neq 0$, then*

$$\gamma_\ell(t) = \begin{cases} O(1) & \text{if } \ell \in [1, L_i] \\ O(\mu_{L_i}) & \text{if } \ell \in [L_i + 1, w] \end{cases} \quad (12)$$

PROOF. See appendix. □

By applying the lemma to (11), we know that, in total, t contributes to $g(1, r_1, ..., r_d)$

$$O(d^2 \mu_{L_i} + w \cdot \mu_{L_i} \cdot d) = O(d^2 \mu_{L_i}).$$

By summing up the contribution of all the tuples, we get:

$$\begin{aligned} &g(1, r_1, ..., r_d) \\ &= O\Big(\sum_{i \in [1,d] \text{ s.t. } L_i \neq 0} \sum_{t \in r_i} d^2 \mu_{L_i} + \sum_{t \in [1,d] \text{ s.t. } L_i = 0} \sum_{t \in r_i} d^2\Big) \\ &= O\Big(\sum_{i \in [1,d] \text{ s.t. } L_i \neq 0} d^2 \mu_{L_i} n_i + \sum_{i=1}^{d} d^2 n_i\Big) \\ &= O\Big(\sum_{\ell=2}^{w} d^2 \mu_{\ell-1} n_{h_\ell} + d^2 \sum_{i=1}^{d} n_i\Big) \end{aligned}$$

where the last equality is due to the definition of L_i.

It remains to bound $\mu_{\ell-1} n_{h_\ell}$ for each $\ell \in [2, w]$. For this purpose, we prove:

LEMMA 6. $\mu_{\ell-1} = O(U d^{\frac{1}{d-1}} / n_{h_\ell})$ *for each* $\ell \in [2, w]$.

PROOF. See appendix. □

The lemma indicates that

$$\begin{aligned} g(1, r_1, ..., r_d) &= O\Big(\sum_{\ell=2}^{w} U d^{2+\frac{1}{d-1}} + d^2 \sum_{i=1}^{d} n_i\Big) \\ &= O\Big(d^{3+\frac{1}{d-1}} U + d^2 \sum_{i=1}^{d} n_i\Big). \end{aligned}$$

Combining the above equation with (1), (10), and Proposition 1, we now complete the whole proof of Theorem 2.

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 3 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. □

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 $n_1 \ge n_2 \ge n_3 \ge M$; otherwise, the algorithm in Lemma 7 already solves the problem in linear I/Os after sorting.

Set:

$$\theta_1 = \sqrt{\frac{n_1 n_3 M}{n_2}}, \text{ and } \theta_2 = \sqrt{\frac{n_2 n_3 M}{n_1}}. \qquad (13)$$

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

$$\begin{aligned} freq(a_1, r_3) &= \text{number of tuples } t \text{ in } r_3 \text{ with } t[A_1] = a_1 \\ freq(a_2, r_3) &= \text{number of tuples } t \text{ in } r_3 \text{ with } t[A_2] = a_2. \end{aligned}$$

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:

1. *Red-red*: $t^*[A_1] \in \Phi_1$ and $t^*[A_2] \in \Phi_2$
2. *Red-blue*: $t^*[A_1] \in \Phi_1$ and $t^*[A_2] \notin \Phi_2$
3. *Blue-red*: $t^*[A_1] \notin \Phi_1$ and $t^*[A_2] \in \Phi_2$
4. *Blue-blue*: $t^*[A_1] \notin \Phi_1$ 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:

$$\begin{aligned} r_3^{red,red} &= \text{set of tuples } t \text{ in } r_3 \text{ s.t. } t[A_1] \in \Phi_1, t[A_2] \in \Phi_2 \\ r_3^{red,blue} &= \text{set of tuples } t \text{ in } r_3 \text{ s.t. } t[A_1] \in \Phi_1, t[A_2] \notin \Phi_2 \\ r_3^{blue,red} &= \text{set of tuples } t \text{ in } r_3 \text{ s.t. } t[A_1] \notin \Phi_1, t[A_2] \in \Phi_2 \\ r_3^{blue,blue} &= \text{set of tuples } t \text{ in } r_3 \text{ s.t. } t[A_1] \notin \Phi_1, t[A_2] \notin \Phi_2 \end{aligned}$$

$$\begin{aligned} r_3^{blue,-} &= r_3^{blue,red} \cup r_3^{blue,blue} \\ r_3^{-,blue} &= r_3^{red,blue} \cup r_3^{blue,blue}. \end{aligned}$$

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:

- $I_1^1, I_2^1, ..., I_{q_1}^1$ are in ascending order.
- For each $j \in [1, q_1]$, $r_3^{blue,-}$ 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:

- $I_1^2, I_2^2, ..., I_{q_2}^2$ are in ascending order.
- For each $j \in [1, q_2]$, $r_3^{-,blue}$ has at most $2\theta_2$ tuples whose A_2-values fall in I_j^2.

We now define several partitions of r_3:

1. For each $a_1 \in \Phi_1$ and $a_2 \in \Phi_2$:
$$\begin{aligned} r_3^{red,red}[a_1, a_2] = {} & \text{the (}\textit{only}\text{) tuple } t \text{ in } r_3^{red,red} \text{ with} \\ & t[A_1] = a_1 \text{ and } t[A_2] = a_2. \end{aligned}$$
2. For each $a_1 \in \Phi_1$ and $j \in [1, q_2]$:
$$\begin{aligned} r_3^{red,blue}[a_1, I_j^2] = {} & \text{set of tuples } t \text{ in } r_3^{red,blue} \text{ with} \\ & t[A_1] = a_1 \text{ and } t[A_2] \text{ in } I_j^2. \end{aligned}$$
3. For each $j \in [1, q_1]$ and $a_2 \in \Phi_2$:
$$\begin{aligned} r_3^{blue,red}[I_j^1, a_2] = {} & \text{set of tuples } t \text{ in } r_3^{blue,red} \text{ with} \\ & t[A_1] \text{ in } I_j^1 \text{ and } t[A_2] = a_2. \end{aligned}$$
4. For each $j_1 \in [1, q_1]$ and $j_2 \in [1, q_2]$:
$$\begin{aligned} r_3^{blue,blue}[I_{j_1}^1, I_{j_2}^2] = {} & \text{set of tuples } t \text{ in } r_3^{blue,blue} \text{ with} \\ & t[A_1] \text{ in } I_j^1 \text{ and } t[A_2] \text{ in } I_j^2. \end{aligned}$$

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

Partitioning r_1 and r_2. Let:

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

We now define several partitions of r_1:

1. For each $a_2 \in \Phi_2$:
$$r_1^{red}[a_2] = \text{set of tuples } t \text{ in } r_1^{red} \text{ with } t[A_2] = a_2.$$
2. For each $j \in [1, q_2]$:
$$r_1^{blue}[I_j^2] = \text{set of tuples } t \text{ in } r_1^{blue} \text{ with } t[A_2] \text{ in } I_j^2.$$

Similarly, we define several partitions of r_2:

1. For each $a_1 \in \Phi_1$:
$$r_2^{red}[a_1] = \text{set of tuples } t \text{ in } r_2^{red} \text{ with } t[A_1] = a_1.$$
2. For each $j \in [1, q_1]$:
$$r_2^{blue}[I_j^1] = \text{set of tuples } t \text{ in } r_2^{blue} \text{ with } t[A_1] \text{ in } I_j^1.$$

It is also fundamental 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.

Emitting Red-Red Tuples. For each $a_1 \in \Phi_1$ and each $a_2 \in \Phi_2$, apply Lemma 7 to emit the result of $r_1^{red}[a_2] \bowtie r_2^{red}[a_1] \bowtie r_3^{red,red}[a_1, a_2]$.

Emitting Red-Blue Tuples. 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 $r_1^{blue}[I_j^2] \bowtie r_2^{red}[a_1] \bowtie r_3^{red,blue}[a_1, I_j^2]$.

Emitting Blue-Red Tuples. 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 $r_1^{red}[a_2] \bowtie r_2^{blue}[I_j^1] \bowtie r_3^{blue,red}[I_j^1, a_2]$.

Emitting Blue-Blue Tuples. For each $j_1 \in [1, q_1]$ and each $j_2 \in [1, q_2]$, apply Lemma 7 to emit the result of $r_1^{blue}[I_{j_2}^2] \bowtie r_2^{blue}[I_{j_1}^1] \bowtie r_3^{blue,blue}[I_{j_1}^1, I_{j_2}^2]$.

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 $r_3^{red,red}[a_1, a_2]$ has only 1 tuple):

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

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

$$\begin{aligned}
&\sum_{a_1, j} O\Big(1 + \frac{|r_1^{blue}[I_j^2]||r_3^{red,blue}[a_1, I_j^2]|}{MB} \\
&+ \frac{|r_1^{blue}[I_j^2]| + |r_2^{red}[a_1]| + |r_3^{red,blue}[a_1, I_j^2]|}{B}\Big). \\
&= O\Big(|\Phi_1| q_2 + \sum_j \frac{|r_1^{blue}[I_j^2]| \sum_{a_1} |r_3^{red,blue}[a_1, I_j^2]|}{MB} \\
&+ \frac{|\Phi_1| \sum_j |r_1^{blue}[I_j^2]|}{B} + \frac{q_2 \sum_{a_1} |r_2^{red}[a_1]|}{B} + \frac{n_3}{B}\Big). \qquad (14)
\end{aligned}$$

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

$$\sum_{a_1} |r_3^{red,blue}[a_1, I_j^2]| \leq 2\theta_2.$$

(14) is thus bounded by:

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

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:

$$\begin{aligned}
&\sum_{j_1, j_2} O\Big(1 + \frac{\big(|r_1^{blue}[I_{j_2}^2]| + |r_2^{blue}[I_{j_1}^1]|\big)|r_3^{blue,blue}[I_{j_1}^1, I_{j_2}^2]|}{MB} \\
&+ \frac{|r_1^{blue}[I_{j_2}^2]| + |r_2^{blue}[I_{j_1}^1]| + |r_3^{blue,blue}[I_{j_1}^1, I_{j_2}^2]|}{B}\Big). \qquad (15)
\end{aligned}$$

Let us analyze each term of (15) in turn. First:

$$\begin{aligned}
&\sum_{j_1, j_2} |r_1^{blue}[I_{j_2}^2]||r_3^{blue,blue}[I_{j_1}^1, I_{j_2}^2]| \\
=\ & \sum_{j_2} |r_1^{blue}[I_{j_2}^2]| \sum_{j_1} |r_3^{blue,blue}[I_{j_1}^1, I_{j_2}^2]| \qquad (16)
\end{aligned}$$

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

$$\sum_{j_1} |r_3^{blue,blue}[I_{j_1}^1, I_{j_2}^2]| \leq 2\theta_2.$$

Therefore:

$$(16) = O\Big(\theta_2 \sum_{j_2} |r_1^{blue}[I_{j_2}^2]|\Big) = O(n_1\theta_2).$$

Symmetrically, we have:

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

Thus, (15) is bounded by:

$$\begin{aligned}
&O\Big(q_1 q_2 + \frac{n_1\theta_2 + n_2\theta_1}{MB} \\
&+ \frac{q_1 \sum_{j_2} |r_1^{blue}[I_{j_2}^2]|}{B} + \frac{q_2 \sum_{j_1} |r_2^{blue}[I_{j_1}^1]|}{B} + \frac{n_3}{B}\Big) \\
=\ & O\Big(q_1 q_2 + \frac{n_1\theta_2 + n_2\theta_1}{MB} + \frac{q_1 n_1}{B} + \frac{q_2 n_2}{B} + \frac{n_3}{B}\Big) \\
=\ & O\Big(\frac{\sqrt{n_1 n_2 n_3}}{B\sqrt{M}} + \frac{n_1}{B}\Big).
\end{aligned}$$

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.

ACKNOWLEDGEMENTS

This work was supported in part by Grants GRF 4168/13 and GRF 142072/14 from HKRGC.

5. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley Publishing Company, 1995.

[2] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *CACM*, 31(9):1116–1127, 1988.

[3] L. Arge, P. Ferragina, R. Grossi, and J. S. Vitter. On sorting strings in external memory (extended abstract). In *STOC*, pages 540–548, 1997.

[4] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. *SIAM J. of Comp.*, 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] P. C. Fischer and D. Tsou. Whether a set of multivalued dependencies implies a join dependency is NP-hard. *SIAM J. of Comp.*, 12(2):259–266, 1983.

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

[8] X. Hu, Y. Tao, and C.-W. Chung. I/O-efficient algorithms on triangle listing and counting. *To appear in ACM TODS*, 2014.

[9] P. C. Kanellakis. On the computational complexity of cardinality constraints in relational databases. *IPL*, 11(2):98–101, 1980.

[10] D. Maier. *The Theory of Relational Databases*. Available Online at http://web.cecs.pdx.edu/∼maier/TheoryBook/TRD.html, 1983.

[11] D. Maier, Y. Sagiv, and M. Yannakakis. On the complexity of testing implications of functional and join dependencies. *JACM*, 28(4):680–695, 1981.

[12] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms: [extended abstract]. In *PODS*, pages 37–48, 2012.

[13] J. Nicolas. Mutual dependencies and some results on undecomposable relations. In *VLDB*, pages 360–367, 1978.

[14] R. Pagh and F. Silvestri. The input/output complexity of triangle enumeration. In *PODS*, pages 224–233, 2014.

APPENDIX

Proof of Lemma 3

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 the tuples t^* in the result of $r_1 \bowtie r_2 \bowtie ... \bowtie r_d$ such that $t^*[A_1] = a$. First, initialize empty sets $S_2, ..., S_d$ in memory. Then, we 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

$$t'[A_j] = t[A_j], \qquad \forall j \in [2, d] \setminus \{i\}. \tag{17}$$

If the answer is no, t is discarded; otherwise, we add it to S_i. Note that the checking happens in memory, and thus, 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 10. *$r_1, S_2, ..., S_d$ 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[4] of the tuple t' in (17). This does not lose any information because we can recover t by resorting to (17) 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 a algorithm of [3] for string sorting in EM. Hence, the overall I/O complexity is as claimed in Theorem 2.

It remains to consider 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.

Proof of Lemma 4

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. The sorted order ensures that if t' exists, then t and t' must appear consecutively during the synchronous scan[5]. 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 $r_1 \bowtie r_2 \bowtie ... \bowtie r_d$. Clearly, $t'[A_i] = t[A_i]$ for all $i \in [1, d] \setminus \{H\}$, and (by definition of point join) $t'[A_H] = a$. 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 $i \in [1, d] \setminus \{H\}$.

Proof of Lemma 5

Let us first understand how t is passed from a call to its descendants in $\mathcal{T}$. Let JOIN$(h_\ell, \rho_1, ..., \rho_d)$ be a level-ℓ call that t participates in. If $h_{\ell+1} \ne i$, then t participates in *at most one* of the call's child nodes in $\mathcal{T}$. Otherwise, t may participate in *all* of the call's child nodes in $\mathcal{T}$.

We first consider the case $L_i = 0$, under which there are two possible scenarios: (i) $i = 1$, or (ii) i is not the axis of any call in $\mathcal{T}$. In neither case will we have a call JOIN$(h_\ell, \rho_1, ..., \rho_d)$ with $h_{\ell+1} = i$. This implies that $\gamma_\ell(t) \le 1$ for all $\ell \in [1, w]$.

Now consider that $L_i \in [1, w-1]$. Let JOIN$(h_\ell, \rho_1, ..., \rho_d)$ be a level-ℓ call that t participates in. If $\ell \ne L_i$, then the call passes t to at most one of its child nodes. If $\ell = L_i$, then by definition of

[4]This address requires only $\lg_2 n_1$ bits by storing an offset.

[5]Note that r_i can have at most one tuple t' that has the same values as t on all attributes in X_i (recall that $t'[A_H]$ is fixed to a by definition of point join).

L_i, we have $i = h_{1+L_i}$. In this scenario, the call may pass t to all its q child nodes where

$$\begin{aligned} q &= O(1 + |\rho_1|/\tau_i) \\ \text{(by (3))} &= O(1 + \tau_{h_{L_i}}/\tau_i) \\ &= O(1 + \tau_{h_{L_i}}/\tau_{h_{1+L_i}}) \\ &= O(\mu_{L_i}). \end{aligned}$$

This implies the equation of $\gamma_\ell(t)$ given in (12).

Proof of Lemma 6

By the definition of $\mu_{\ell-1}$, it suffices to show that $\tau_{h_{\ell-1}}/\tau_{h_\ell} = O(Ud^{\frac{1}{d-1}}/n_{h_\ell})$. (2) implies that

$$\frac{\tau_{h_{\ell-1}}}{\tau_{h_\ell}} = \frac{(Ud^{\frac{1}{d-1}})^{h_\ell - h_{\ell-1}}}{\prod_{j=1+h_{\ell-1}}^{h_\ell} n_j}. \tag{18}$$

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

$$(18) = \frac{Ud^{\frac{1}{d-1}}}{n_{h_\ell}}.$$

For the case where $h_\ell > 1 + h_{\ell-1}$, the definition of h_ℓ indicates that

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

otherwise, h_ℓ would not be the smallest integer in $[1 + h_{\ell-1}, d]$ satisfying $\tau_{h_\ell} < \tau_{h_{\ell-1}}/2$. Hence,

$$(18) \le 2 \cdot \frac{Ud^{\frac{1}{d-1}}}{n_{h_\ell}},$$

which completes the proof.