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

\[ J = \Join[R_1, R_2, ..., R_m] \]

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

\[ r = \pi_{R_1}(r) \Join \pi_{R_2}(r) \Join ... \Join \pi_{R_m}(r) \]

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

**Problem 1.** [\( \lambda \)-JD Testing] Given a relation \( r \) and a join dependency \( J \) of arity at most \( \lambda \) 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 \( \log_2 y = \max \{1, \log_2 y\} \), and will
describe all logarithms using \( \log_2 y \). 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 \( R = \{A_1, A_2, \ldots, A_d\} \) 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 \( r_1, r_2, \ldots, r_d \) be \( d \) relations such that
\( r_i \ (1 \leq i \leq d) \) has schema \( R_i \). Then, the natural join
\( r_1 \bowtie r_2 \bowtie \ldots \bowtie r_d \) 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 \( \text{emitt}(\cdot) \) which requires \( O(1) \)
words to store. The parameter of the routine is a tuple of \( d \) values
\( (a_1, \ldots, a_d) \) 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 \( r_1, \ldots, r_d \)
defined earlier where \( d \leq M/2 \), we want to invoke \( \text{emitt}(t) \) once
and exactly once for each tuple \( t \in r_1 \bowtie r_2 \bowtie \ldots \bowtie r_d \).

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 \( \text{emitt}(\cdot) \) that occupies \( O(1) \) words. This time, given a
triangle \( \Delta \) as its parameter, the routine sends out \( \Delta \) to an outbound
socket with no I/O cost (this implies that all the 3 edges of \( \Delta \) 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 \( \text{emitt}(\Delta) \) once and exactly once
for each triangle \( \Delta \) 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 \( \lambda \)-JD
testing (Problem 1) is NP-hard if \( \lambda = d - \alpha(d) \); recall that \( d \) is
the number of attributes in the input relation \( r \). Maier, Sagiv, and
Yannakakis [11] gave a stronger proof showing that \( \lambda \)-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 \( \bowtie [R_1, R_2, \ldots, R_m] \) on \( r \), when one of
\( R_1, \ldots, R_m \) 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
\( R_1, \ldots, R_m \) 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 \( \lambda_1 \)-JD testing
problem may only be easier than \( \lambda_2 \)-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 \( \Sigma \) 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 \( \Sigma \) contains one join dependency and a set of
functional dependencies [5, 11], (ii) when \( \Sigma \) is a set of multi-valued
dependencies [6], and (iii) when \( \Sigma \) 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 \( \lambda \)-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 \( R = \{A_1, A_2, \ldots, A_d\} \).
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 \( r = r_1 \bowtie r_2 \bowtie \ldots \bowtie r_d \). In fact, since it is always true that \( r \subseteq r_1 \bowtie r_2 \bowtie \ldots \bowtie r_d \),
Problem 2 has an answer yes if and only if \( r \bowtie r_2 \bowtie \ldots \bowtie r_d \) returns exactly \( |r| \)
result tuples.

Therefore, Problem 2 boils down to evaluating the result size of
the LW join \( r_1 \bowtie r_2 \bowtie \ldots \bowtie r_d \). Atserias, Grohe, and Marx
[4] showed that the result size can be as large as \( (n_1n_2 \ldots n_d)^2 \sum_{i=1}^{d} n_i \) \( \in \cdot \)
where \( n_i = |r_i| \) for each \( i \in [1, d] \). They also gave a RAM
algorithm to compute the join result in \( O(d^2 \cdot (n_1n_2 \ldots n_d)^2 \sum_{i=1}^{d} n_i) \) time. Since apparently \( n_i \leq n = \left| r \right| \) \( (1 \leq i \leq d) \),
it follows that their algorithm has running time \( O(d^2 \cdot n^{2\alpha(d-1)}) \cdot 
\sum_{i=1}^{d} n_i \) \( \cdot \)
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
\( O(d^2 \cdot (n_1n_2 \ldots n_d)^2 \sum_{i=1}^{d} n_i) \) 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 \(O(d^2 \cdot (n_1 n_2 \ldots n_d)^{2\lambda} + d^2 \sum_{i=1}^{d} n_i)\) I/Os. When \(d\) is small, this may be even worse than a naive generalized blocked-nested loop, whose I/O complexity for \(d = 0(1)\) is \(O(n_1 n_2 \ldots n_d / (M^{d-1} B))\) 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 \(O(|E|^{1.5} / (\sqrt{MB}))\) 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 \(O\left(\frac{|E|^{1.5}}{\sqrt{MB}} \log M/B\right)\) I/Os. An I/O lower bound of \(O(|E|^{1.5} / (\sqrt{MB}))\) 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-JD testing is NP-hard). Our second main result is an I/O-efficient algorithm for EM enumeration (Problem 3). Let \(r_1, r_2, \ldots, r_d\) 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:

\[
O\left(\text{sort}\left(\frac{d^2 + o(1) \left(\Pi_{i=1}^{d} n_i\right)^{\lambda}}{M} + \frac{d^2}{\sum_{i=1}^{d} n_i}\right)\right),
\]

where function \(\text{sort}(x)\) equals \((x/B)\log_{M/B}(x/B)\).

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 \(\Omega(\frac{d}{M} \left(\Pi_{i=1}^{d} n_i\right)^{1/(d-1)})\) 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 \(O\left(\frac{1}{B} \sqrt{\max(n_1, n_2 + n_3)}\right)\).

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 \(\tau(R)\) be the input relation to the JD existence testing problem, where \(R = \{A_1, \ldots, A_d\}\). For each \(i \in [1, d]\), define \(R_i = R \setminus \{A_i\}\), and \(n_i\) as the number of tuples in \(r_i\). 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 \(\text{sort}(|E|) = O(|E|^{1.5} / (\sqrt{MB}))\):

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

Our triangle enumeration algorithm is deterministic, and strictly improves that of [14] by a factor of \(O(\log_{M/B}(|E|/B))\). 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 \(\ell\) in \(G\) is a sequence of \(\ell\) vertices \(v_1, v_2, \ldots, v_\ell\) 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: \(\{A_1, A_2, \ldots, A_n\}\). We will create \(\binom{n}{2}\) relations. Specifically, for each pair of \(i, j\) such that \(1 \leq i < j \leq n\), 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 \(j > i + 2\):** \(r_{i,j}\) contains \(n(n - 1)\) tuples \((x, y)\), for all possible integers \(x, y\), such that \(x \neq y\), and \(1 \leq x, y \leq n\).

In general, the total number of tuples in the \(r_{i,j}\) of all possible \(i, j\) is \(O(nm + n^2) = O(n^3)\).

Define:

\[
\text{CLIQUE} = \text{the output of the natural join of all } r_{i,j} (1 \leq i < j \leq n).
\]

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

**Lemma 1.** \(G\) has a Hamiltonian path if and only if \(\text{CLIQUE}\) is not empty.

**Proof.** **Direction If:** Assuming that \(\text{CLIQUE}\) is not empty, next we show that \(G\) has a Hamiltonian path. Let \((id(v_1), id(v_2), \ldots, id(v_\ell))\) be an arbitrary tuple in \(\text{CLIQUE}\). It follows that:

\(^1\)Recall that a graph is simple if it has at most one edge between any two vertices.
Assuming that we show that $C$ in LIQUE not satisfy against Fact 3.

**Direction Only-If.** Assuming that $G$ has a Hamiltonian path, next we show that CLIQUE is not empty. Let $r_1, r_2, \ldots, r_n$ be any Hamiltonian path in $G$. It is easy to verify that $(id(v_1), id(v_2), \ldots, id(v_n))$ 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:

$$J = \{\exists R_{i,j}, \forall v_i \text{ s.t. } 1 \leq i < j \leq n\}.$$

For instance, for $n = 3$, $J = \{R_{1,2}, R_{1,3}, R_{2,3}\}$. 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 $m \leq n^3$).

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 $t = (a_1, a_2, \ldots, a_n)$. Then, $t'$ is determined as follows:

- $t'[A_i] = a_i$ (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 relation $r^*$.

Since (as mentioned before) there are $O(n^3)$ tuples in the $r_{i,j}$ of all $i, j$, we know that $r^*$ has $O(n^3)$ tuples, and hence, can be built in $O(n^3)$ 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 $(a_1, a_2, \ldots, a_n)$ is a tuple in CLIQUE. Hence, $(a_1, a_2, \ldots, a_n)$ is a tuple in $r^*$ 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 follows that $(a_1, a_2, \ldots, a_n)$ is a tuple in CLIQUE$^*$. However, by Fact 1, $(a_1, a_2, \ldots, a_n)$ 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 $(a^*_1, a^*_2, \ldots, a^*_n)$ be a tuple in $\text{CLIQUE}^*$ but not in $r^*$. We distinguish two cases:

- **Case 1:** none of $a^*_1, \ldots, a^*_n$ 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, $(a^*_1, a^*_2, \ldots, a^*_n)$ 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 $(a_1, a_2, \ldots, a_n)$, thus contradicting the assumption that $(a^*_1, a^*_2, \ldots, a^*_n)$ is not in $r^*$, which will then complete the proof.

Consider any $i$ such that $1 \leq i < k$. That $(a^*_1, a^*_2, \ldots, a^*_n)$ 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 $t^*[A_i] = a_k^*$. 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 $(a_1^*, a_2^*, \ldots, a_n^*)$ 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 $R = \{A_1, A_2, \ldots, A_d\}$. For each $i \in [1, d]$, let $R_i = R\{A_i\}$. We are given relations $r_1, r_2, \ldots, r_d$ 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 $r_1 \bowtie r_2 \bowtie \ldots \bowtie r_d$, 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 $r_1, \ldots, r_d$ 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 $r_1 \bowtie r_2 \bowtie \ldots \bowtie r_d$ a small join.

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

**Proof.** See appendix.

The second scenario takes a bit more efforts to explain. In addition to $r_1, \ldots, r_d$, 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 \( r_1, r_H, r_{H+1}, \ldots, r_d \) (recall that \( r_H \) does not have \( A_H \)). In such a case, we call \( r_1 \bowtie r_2 \bowtie \cdots \bowtie r_d \) a point join.

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

**Proof.** See appendix. □

We will denote the algorithm in the above lemma as \( \text{PtJOIN}(H, a, r_1, r_2, \ldots, r_d) \).

### 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 \( n_1 \leq 2M/d \), simply apply Lemma 3 because this is a small-join scenario.

Define:

\[
U = \left( \prod_{i=1}^{d} n_i \right)^{1/M}
\]

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

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

Our general algorithm is a recursive procedure \( \text{JOIN}(h, \rho_1, \ldots, \rho_d) \), which has three requirements:

- \( h \) is an integer in \([1, d]\);
- Each \( \rho_i \) (\( 1 \leq i \leq d \)) is a subset of the tuples in \( r_i \);
- The size of \( \rho_i \) satisfies:

\[
|\rho_i| \leq \tau_h.
\]

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

#### 3.2.1 Case \( \tau_h \leq 2M/d \)

In this case, by the requirements of \( \text{JOIN}(h, \rho_1, \ldots, \rho_d) \), it holds that \( |\rho_1| \leq \tau_h = O(M/d) \). 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 \( a \in \text{dom}(A_H) \), we define:

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

Now we introduce:

\[
\Phi = \{ a \in \text{dom}(A_H) \mid \text{freq}(a) > \tau_h/2 \}.
\]

Let \( t^* \) be a result tuple of \( \rho_1 \bowtie \cdots \bowtie \rho_d \). 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 \( \rho_i \) into:

\[
\rho_i^{\text{red}} = \{ \text{tuples } t \in \rho_i \mid t[A_H] \in \Phi \}
\]

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

To emit red tuples, it suffices to consider \( \rho_1^{\text{red}}, \ldots, \rho_{H-1}^{\text{red}}, \rho_H^{\text{red}}, \rho_{H+1}^{\text{red}}, \ldots, \rho_d^{\text{red}} \). Likewise, to emit blue tuples, it suffices to consider \( \rho_1^{\text{blue}}, \ldots, \rho_{H-1}^{\text{blue}}, \rho_H^{\text{blue}}, \rho_{H+1}^{\text{blue}}, \ldots, \rho_d^{\text{blue}} \). Next, we will elaborate on how to do so.

**Remark.** The set \( \Phi \), as well as \( \rho_i^{\text{red}} \) and \( \rho_i^{\text{blue}} \) for each \( i \in [1, d] \setminus \{H\} \), can be produced by sorting each \( \rho_i \) 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 \( O(d + \text{sort}(d \sum_{i \in \{1, d\} \setminus \{H\}} |\rho_i|)) \) 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 \( i \in [1, d] \setminus \{H\} \):

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

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

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

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

- \( I_1, I_2, \ldots, I_q \) are in ascending order;
- For each \( j \in [1, q] \), define:

\[
\rho_1^{\text{blue}}[I_j] = \{ \text{tuples } t \in \rho_1^{\text{blue}} \text{ whose } A_H-\text{values fall in } I_j \}
\]

If \( j < q \), we require \( \tau_H/2 \leq |\rho_1^{\text{blue}}[I_j]| \leq \tau_H \). Regarding \( \rho_1^{\text{blue}}[I_q] \), we require \( 1 \leq |\rho_1^{\text{blue}}[I_q]| \leq \tau_H \).

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

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

\[
\rho_i^{\text{blue}}[I_j] = \{ \text{tuples } t \in \rho_i^{\text{blue}} \text{ whose } A_H-\text{values fall in } I_j \}
\]

Because \( \rho_1^{\text{blue}} \) has been sorted by \( A_H \), all the \( \rho_1^{\text{blue}}[I_1], \rho_1^{\text{blue}}[I_2], \ldots, \rho_1^{\text{blue}}[I_q] \) can be obtained by scanning synchronously \( \rho_1^{\text{blue}} \) and \( \{I_1, \ldots, I_q\} \) once.

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

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

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

### 3.3 Analysis

Define a sequence of integers as follows:

- \( h_1 = 1 \);
- After \( h_1 \) has been defined (\( i \geq 1 \)):
  - 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 \( \text{JOIN} \) procedure with \( \text{JOIN}(1, r_1, \ldots, r_d) \), which recursively makes

\[
\text{An interval } [x, y] \text{ precedes another } [x', y'] \text{ if } y < x'.
\]
f(ℓ, ρ₁, ..., ρₙ) = \begin{cases} O(d) & \text{if } ℓ = w \\ O(d \cdot μ_ℓ) + \sum_{j=1}^{q} f(ℓ + 1, ρ₁^{blue}[I_j], ..., ρ_ℓ^{blue}[I_j], ρ_{h_{ℓ+1}-1}[I_j], ρ_{h_{ℓ+1}}[I_j], ..., ρ_d^{blue}[I_j]) & \text{if } ℓ < w \end{cases}

g(ℓ, ρ₁, ..., ρₙ) = \begin{cases} d \cdot \sum_{i=1}^{d} |ρ_i| & \text{if } ℓ = w \\ d^2 μ_ℓ |ρ_{h_{ℓ+1}}| + d \cdot \sum_{i=1}^{d} |ρ_i| + \sum_{j=1}^{q} g(ℓ + 1, ρ₁^{blue}[I_j], ..., ρ_ℓ^{blue}[I_j], ρ_{h_{ℓ+1}-1}[I_j], ρ_{h_{ℓ+1}}[I_j], ..., ρ_d^{blue}[I_j]) & \text{if } ℓ < w \end{cases}

Figure 1: Definitions of f(h, ρ₁, ..., ρₙ) and g(h, ρ₁, ..., ρₙ)

Recall that q is the number of disjoint intervals that \text{JOIN}(h₁, ρ₁, ..., ρₙ) uses to divide \text{dom}(A₁) for blue tuple emission (see Section 3.2).

The rest of the subsection is devoted to solving this non-conventional recurrence. Let functions f(ℓ, ρ₁, ..., ρₙ) and g(ℓ, ρ₁, ..., ρₙ) be as defined in Figure 1. The following proposition is fundamental:

**Proposition 1.** \( \text{cost}(ℓ, ρ₁, ..., ρₙ) = f(ℓ, ρ₁, ..., ρₙ) + O(\text{sort}(g(ℓ, ρ₁, ..., ρₙ))) \).

**Proof.** By the convexity of function \( \text{sort}(x) \).

To prove Theorem 2, our target is to give an upper bound on \( \text{cost}(1, r₁, ..., rₙ) = f(1, r₁, ..., rₙ) + O(\text{sort}(g(1, r₁, ..., rₙ))) \).

### 3.3.1 Bounding \( f(1, r₁, ..., rₙ) \)

Define \( m_ℓ \) as the total number of level-ℓ calls in \( T \). Each level-ℓ call contributes \( O(d \cdot μ_ℓ) \) I/Os to \( f(1, r₁, ..., rₙ) \) (see Figure 1). Hence:

\[
f(1, r₁, ..., rₙ) = \sum_{ℓ=1}^{w} O(m_ℓ \cdot d \cdot μ_ℓ).
\]

We say that a level-ℓ call \( \text{JOIN}(h₁, ρ₁, ..., ρₙ) \) underflows if \( |ρ₁| < τₙ₁/2 \); otherwise, we say that it is ordinary. Consider all the calls \( \text{JOIN}(h₁, ρ₁, ..., ρₙ) \) at level ℓ. The sets \( ρ₁ \) in the first parameters of those calls are disjoint. Hence, there can be at most \( O(n₁/τₙ₁) \) ordinary calls at level ℓ. Moreover, if \( ℓ < w \), then a level-ℓ call creates at most one underflowing call at level ℓ + 1. These facts indicate that, for each \( ℓ \in [2, w] \):

\[
m_ℓ = O\left(\frac{n₁}{τₙ₁} + \frac{n₁}{τₙ₁}ight) = O\left(\sum_{i=1}^{ℓ} \frac{n₁}{τₙ_i}\right) = O\left(\frac{n₁}{τₙ₁}\right),
\]

where the second equality used \( m₁ = n₁/τₙ₁ \), and the last equality used the fact that \( τₙ_i > 2τₙ_{i+1} \) for every \( i \in [1, w - 1] \). Applying \( τₙₘ = M/d \), we get from (9):

\[
mₘ = O(\frac{n₁}{τₙ₁}).
\]

Moreover, for each \( ℓ \in [1, w - 1] \):

\[
m₄ℓμ₄ = O\left(\frac{n₁}{τₙ₁} \cdot \frac{2τₙ₄}{τₙ₄}\right) = O\left(\frac{n₁}{τₙ₄}\right).
\]

Here define a boundary dummy \( μ₄w = 1 \).
We can now derive from (8):

\[
 f(1, r_1, ..., r_d) = O \left( \frac{d^2 n_1}{M} + \sum_{\ell=1}^{w-1} \frac{d \cdot n_1}{\tau_{n_{\ell+1}}} \right) 
\]

\[
 = O \left( \frac{d^2 n_1}{M} + \frac{d n_1}{\tau_{n_w}} \right) 
\]

\[
 = O \left( \frac{d^2 n_1}{M} \right). \quad (10)
\]

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

Figure 1 shows that, in \( T \), each level-\( \ell \) (\( \ell < w \)) call JOIN(\( h_\ell, p_\ell, L_\ell ..., p_d \)) contributes \( d^2 \mu |p_{\ell+1}| + d \sum_{i=1}^{d} |p_i| \) to \( g(1, r_1, ..., r_d) \). We can amortize the contribution onto the tuples in \( p_1, ..., p_d \), such that:

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

Similarly, for every level-\( w \) call JOIN(\( h_w, p_1, p_2, ..., p_d \)), each tuple in \( p_1, ..., p_w \) 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 \( T \) has axis \( i \), then \( L_i = 0 \);
- Otherwise, suppose that the level-\( \ell \) 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-\( \ell \) call (\( 1 \leq \ell \leq w \)) JOIN(\( h_\ell, p_\ell, ..., p_d \)) in \( T \). We say that \( t \) participates in the call if \( t \in p_\ell \). 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 \( \gamma(t) \) the number of level-\( \ell \) calls that \( t \) participates in; specially, define \( \gamma(t) = 0 \). Then, the sequence \( L_1, L_2, ..., L_d \) defined earlier allows us to represent concisely the total contribution of \( t \) as

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

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

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

Proof. See appendix. \( \Box \)

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

\[
 O(d^2 \mu_\ell + w \cdot \mu_\ell \cdot d) = O(d^2 \mu_\ell). 
\]

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

\[
 g(1, r_1, ..., r_d) = O \left( \sum_{i=1}^{w-1} d^2 \mu_i \sum_{\ell=1}^{d} |p_i| + \sum_{\ell=2}^{w} d \right) 
\]

\[
 = O \left( \sum_{i=1}^{w} d^2 \mu_i n_i + \sum_{i=1}^{d} n_i \right) 
\]

\[
 = O \left( \sum_{\ell=2}^{w} d^2 \mu_{\ell-1} n_{h_\ell} + d^2 \sum_{i=1}^{d} n_i \right) 
\]

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

Proof. See appendix. \( \Box \)

The lemma indicates that

\[
 g(1, r_1, ..., r_d) = O \left( \sum_{\ell=2}^{w} U d^{1/\ell} + d^2 \sum_{i=1}^{d} n_i \right) 
\]

\[
 = O \left( d^{1/2} U + d^2 \sum_{i=1}^{d} n_i \right). 
\]

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: \( r_1(A_2, A_3), r_2(A_1, 3), \) 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 \( O(1 + \frac{n_1 \cdot n_2}{M^2}) + \frac{1}{M} \sum_{i=1}^{3} n_i \) 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. \( \Box \)

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 \( O(1 + \frac{n_1 \cdot n_2}{M^2} + \frac{1}{M^3} \sum_{i=1}^{3} n_i) \) I/Os.
Proof. We first obtain $r' = (A_1, A_2, A_3) = r_1 \otimes r_2$, 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 $|r'| \leq n_1$. 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 $O((n_1 \cdot n_2)/B)$ I/Os.

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

Symmetrically, we call $r_1 \otimes r_2 \otimes r_3$ 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 $O(1 + \frac{n_3}{MB} + \frac{1}{B} \sum_{\delta=1}^{n_1} n_1)$ 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 $n_1 \geq n_2 \geq n_3 \geq M$; otherwise, the algorithm in Lemma 7 already solves the problem in linear I/Os after sorting.

Set:

\[ \theta_1 = \sqrt{\frac{n_1 \cdot n_3 \cdot M}{n_2}}, \quad \theta_2 = \sqrt{\frac{n_2 \cdot n_3 \cdot M}{n_1}}. \]  

(13)

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

\[
\text{freq}(a_1, r_3) = \text{number of tuples } t \text{ in } r_3 \text{ with } t[A_1] = a_1
\]

\[
\text{freq}(a_2, r_3) = \text{number of tuples } t \text{ in } r_3 \text{ with } t[A_2] = a_2.
\]

Now we introduce:

\[
\Phi_1 = \{a_1 \in \text{dom}(A_1) \mid \text{freq}(a_1, r_3) > \theta_1\}
\]

\[
\Phi_2 = \{a_2 \in \text{dom}(A_2) \mid \text{freq}(a_2, r_3) > \theta_2\}.
\]

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

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

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:

\[
\text{red-red} \quad r^3_{\text{red,red}} = \text{set of tuples } t \text{ in } r_3 \text{ s.t. } t[A_1] \in \Phi_1, t[A_2] \in \Phi_2
\]

\[
\text{red-blue} \quad r^3_{\text{red,blue}} = \text{set of tuples } t \text{ in } r_3 \text{ s.t. } t[A_1] \in \Phi_1, t[A_2] \notin \Phi_2
\]

\[
\text{blue-red} \quad r^3_{\text{blue,red}} = \text{set of tuples } t \text{ in } r_3 \text{ s.t. } t[A_1] \notin \Phi_1, t[A_2] \in \Phi_2
\]

\[
\text{blue-blue} \quad r^3_{\text{blue,blue}} = \text{set of tuples } t \text{ in } r_3 \text{ s.t. } t[A_1] \notin \Phi_1, t[A_2] \notin \Phi_2
\]

Divide $\text{dom}(A_1)$ into $q_1 = O(1 + n_3/\theta_1)$ disjoint intervals $I_1^1, I_1^2, \ldots, I_1^{q_1}$ with the following properties:

- $I_1^1, I_1^2, \ldots, I_1^{q_1}$ are in ascending order.
- For each $j \in [1, q_1]$, $r^3_{\text{blue,red}}$ has at most $2\theta_1$ tuples whose $A_1$-values fall in $I_1^j$.

Similarly, we divide $\text{dom}(A_2)$ into $q_2 = O(1 + n_3/\theta_2)$ disjoint intervals $I_2^1, I_2^2, \ldots, I_2^{q_2}$ with the following properties:

- $I_2^1, I_2^2, \ldots, I_2^{q_2}$ are in ascending order.
- For each $j \in [1, q_2]$, $r^3_{\text{blue,blue}}$ has at most $2\theta_2$ tuples whose $A_2$-values fall in $I_2^j$.

We now define several partitions of $r_3$:

1. For each $a_1 \in \Phi_1$ and $a_2 \in \Phi_2$:

\[
r^3_{\text{red,red}}[a_1, a_2] = \text{the (only) tuple } t \text{ in } r^{3,\text{red,red}} \text{ with } t[A_1] = a_1 \text{ and } t[A_2] = a_2.
\]

2. For each $a_1 \in \Phi_1$ and $j \in [1, q_2]$:

\[
r^3_{\text{red,blue}}[a_1, j] = \text{set of tuples } t \text{ in } r^{3,\text{blue,red}} \text{ with } t[A_1] = a_1 \text{ and } t[A_2] \in I_2^j.
\]

3. For each $j \in [1, q_1]$ and $a_2 \in \Phi_2$:

\[
r^3_{\text{blue,red}}[j, a_2] = \text{set of tuples } t \text{ in } r^{3,\text{blue,red}} \text{ with } t[A_1] \in I_1^j \text{ and } t[A_2] = a_2.
\]

4. For each $j \in [1, q_1]$ and $j_2 \in [1, q_2]$:

\[
r^3_{\text{blue,blue}}[j, j_2] = \text{set of tuples } t \text{ in } r^{3,\text{blue,blue}} \text{ with } t[A_1] \in I_1^j \text{ and } t[A_2] \in I_2^{j_2}.
\]

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

Partitioning $r_1$ and $r_2$. Let:

\[
r^1_{\text{red}} = \text{set of tuples } t \text{ in } r_1 \text{ s.t. } t[A_2] \in \Phi_2
\]

\[
r^1_{\text{blue}} = \text{set of tuples } t \text{ in } r_1 \text{ s.t. } t[A_2] \notin \Phi_2
\]

\[
r^2_{\text{red}} = \text{set of tuples } t \text{ in } r_2 \text{ s.t. } t[A_1] \in \Phi_1
\]

\[
r^2_{\text{blue}} = \text{set of tuples } t \text{ in } r_2 \text{ s.t. } t[A_1] \notin \Phi_1
\]

We now define several partitions of $r_1$:

1. For each $a_2 \in \Phi_2$:

\[
r^1_{\text{red}}[a_2] = \text{set of tuples } t \text{ in } r^1_{\text{red}} \text{ with } t[A_2] = a_2.
\]

2. For each $j \in [1, q_2]$:

\[
r^2_{\text{blue}}[j] = \text{set of tuples } t \text{ in } r^2_{\text{blue}} \text{ with } t[A_2] \in I_2^j.
\]

Similarly, we define several partitions of $r_2$:

1. For each $a_1 \in \Phi_1$:

\[
r^2_{\text{red}}[a_1] = \text{set of tuples } t \text{ in } r^2_{\text{red}} \text{ with } t[A_1] = a_1.
\]

2. For each $j \in [1, q_1]$:

\[
r^2_{\text{blue}}[j] = \text{set of tuples } t \text{ in } r^2_{\text{blue}} \text{ with } t[A_1] \in I_1^j.
\]
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 $r_1^{blue}[I_j^2] \bowtie r_2^{blue}[I_j^1]$.

**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 $r_1^{red}[a_2] \bowtie r_2^{blue}[I_j^1] \bowtie r_3^{blue,red}[a_1, I_j^2]$.

**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 $r_1^{blue}[I_j^2] \bowtie r_2^{blue}[I_j^1] \bowtie r_3^{blue,blue}[I_{j_1}, I_{j_2}^2]$.

### 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

$$|\Phi_1| \leq \frac{n_3}{\theta_1} = \sqrt{\frac{n_2 n_3}{n_1 M}}$$

$$|\Phi_2| \geq \frac{n_3}{\theta_2} = \sqrt{\frac{n_1 n_2}{n_2 M}}$$

$$q_1 = O\left(1 + \frac{n_3}{\theta_1}\right) = O\left(1 + \sqrt{\frac{n_2 n_3}{n_1 M}}\right)$$

$$q_2 = O\left(1 + \frac{n_3}{\theta_2}\right) = O\left(1 + \sqrt{\frac{n_1 n_2}{n_2 M}}\right).$$

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):

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

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

$$\sum_{a_1, j} O\left(1 + \frac{|r_1^{blue}[I_j^2]|}{MB} + \frac{|r_2^{blue}[I_j^1]|}{B} + \frac{|r_3^{red,blue}[a_1, I_j^2]|}{B}\right),$$

$$= O\left(\frac{n_1 |\Phi_2|}{M} + \sum_{a_1} \frac{|r_1^{blue}[I_j^2]| |\Phi_1|}{B} + \frac{|r_2^{blue}[I_j^1]|}{B} + \frac{|r_3^{red,blue}[a_1, I_j^2]|}{B}\right) = O\left(\frac{n_1 n_2 n_3}{M B} + \frac{n_3 |\Phi_2|}{B}\right).$$

(14) is thus bounded by:

$$O\left(\frac{n_1 n_3}{M} + \sum_{j} \frac{|r_1^{blue}[I_j^2]|}{MB} + \frac{|\Phi_1| n_1}{B} + \frac{q_1 n_2}{B} + \frac{n_3}{B}\right)$$

$$= O\left(\frac{n_1 n_2}{MB} + \frac{|\Phi_1| n_1}{B} + \frac{q_2 n_2}{B} + \frac{n_3}{B}\right) = O\left(\sqrt{n_1 n_2 n_3} B \sqrt{M}\right).$$

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

$$\sum_{j_1, j_2} O\left(1 + \frac{|r_1^{blue}[I_{j_1}^2]|}{MB} + \frac{|r_2^{blue}[I_{j_1}^1]|}{B} + \frac{|r_3^{blue,blue}[I_{j_1}, I_{j_2}^2]|}{B}\right).$$

(15)

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

$$\sum_{j_1, j_2} |r_1^{blue}[I_{j_1}^2]| |r_3^{blue,blue}[I_{j_1}, I_{j_2}^2]|$$

$$= \sum_{j_2} |r_1^{blue}[I_{j_2}^2]| \sum_{j_1} |r_3^{blue,blue}[I_{j_1}, I_{j_2}^2]|$$

(16)

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

gives the number of tuples in $r_3^{blue,blue}$ whose $A_3$-values fall in $I_{j_2}^2$. By the way $I_{j_1}^1$, ..., $I_{j_2}^2$ are constructed, we know:

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

Therefore:

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

Symmetrically, we have:

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

Thus, (15) is bounded by:

$$O\left(q_1 q_2 + \frac{n_1 n_2 \theta_1 + n_2 \theta_1}{MB} + \frac{q_1 \sum_{j_2} |r_1^{blue}[I_{j_2}^2]| + q_2 \sum_{j_1} |r_2^{blue}[I_{j_1}^1]|}{B} + \frac{n_3}{B}\right)$$

$$= O\left(q_1 q_2 + \frac{n_1 n_2 \theta_1 + n_2 \theta_1}{MB} + \frac{q_1 n_1}{B} + \frac{q_2 n_2}{B} + \frac{n_3}{B}\right)$$

$$= O\left(\frac{n_1 n_2 n_3}{B \sqrt{M}} + \frac{n_1}{B}\right).$$

As already mentioned in Section 4.2, the partitioning phase requires $O(sort(\sum_{i=1}^{n} n_i))$ 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


Without loss of generality, suppose that \( r \). Then, we process each tuple \( t \) emit all the tuples above tuple emission incurs no I/Os due to the following lemma. That the checking happens in memory, and thus, entails no I/O.

Let us first understand how \( r \) be kept in memory throughout the entire algorithm. With \( r \) fit in memory, we merge all the tuples of \( A \) in memory, we use the procedure below to emit all the result tuples produced from each of the subsets.

Proof of Lemma 3

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. If \( t' \) exists, \( t \) is kept in \( r_H \); otherwise, we discard \( t \) from \( r_H \). (\( t \) cannot produce any tuple in \( r_1 \bowtie r_2 \bowtie \ldots \bowtie r_d \)).

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 \( r_1 \bowtie r_2 \bowtie \ldots \bowtie r_d \). Clearly, \( t'[A_i] = \{t[A_i] \setminus \{i\} \} \), 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 \( T \). Let \( \text{JOIN}(h_{\ell}, \rho_1, \ldots, \rho_d) \) be a level-\( \ell \) call that \( t \) participates in. If \( h_{\ell+1} \neq i \), then \( t \) participates in \( (a) \) call of the call’s child nodes in \( T \). Otherwise, \( t \) may participate in all of the call’s child nodes in \( T \).

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 \( T \). In neither case will we have a call \( \text{JOIN}(h_{\ell}, \rho_1, \ldots, \rho_d) \) with \( h_{\ell+1} = i \). This implies that \( \gamma(\ell, i) \leq 1 \) for all \( i \in [1, w] \).

Now consider that \( L_i \in [1, w - 1] \). Let \( \text{JOIN}(h_{\ell}, \rho_1, \ldots, \rho_d) \) be a level-\( \ell \) call that \( t \) participates in. If \( \ell \neq L_i \), then the call passes \( t \) to at most one of its child nodes. If \( \ell = L_i \), then by definition of
If $h_\ell = 1 + h_{\ell-1}$, then
\begin{equation}
\frac{\tau_{h_{\ell-1}}}{\tau_{h_\ell}} = \frac{Ud^{\frac{1}{h_\ell}}}{{\frac{1}{h_\ell}}}
\end{equation}
For the case where $h_\ell > 1 + h_{\ell-1}$, the definition of $h_\ell$ indicates that
\begin{equation}
\frac{\tau_{h_{\ell-1}}}{\tau_{h_\ell}} = \frac{(Ud^{\frac{1}{h_\ell}})^{h_{\ell-1} - h_\ell - 1}}{\prod_{j=1}^{h_{\ell-1}} n_j} \leq 2;
\end{equation}
otherwise, $h_\ell$ would not be the smallest integer in $[1 + h_{\ell-1}, d]$ satisfying $\tau_{h_\ell} < \tau_{h_{\ell-1}}/2$. Hence,
\begin{equation}
\frac{\tau_{h_{\ell-1}}}{\tau_{h_\ell}} \leq 2 \cdot \frac{Ud^{\frac{1}{h_{\ell-1}}}}{n_{h_\ell}},
\end{equation}
which completes the proof.

If $h_1 = L_1$, we have $i = h_1 + L_1$. In this scenario, the call may pass $t$ to all its $q$ child nodes where
\begin{align*}
q &= O(1 + |p_1|/\tau_1) \\
(\text{by (3)}) &= O(1 + \tau_{h_1}/\tau_1) \\
&= O(1 + \tau_{h_1}/\tau_{h_1 + L_1}) \\
&= O(\mu_{L_1}).
\end{align*}
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 $\tau_{h_{\ell-1}}/\tau_{h_\ell} = O(Ud^{\frac{1}{h_\ell}}/\tau_{h_\ell})$. (2) implies that
\begin{equation}
\frac{\tau_{h_{\ell-1}}}{\tau_{h_\ell}} = \frac{(Ud^{\frac{1}{h_\ell}})^{h_{\ell-1} - h_\ell - 1}}{\prod_{j=1}^{h_{\ell-1}} n_j}.
\end{equation}

If $h_\ell = 1 + h_{\ell-1}$, then
\begin{equation}
\frac{\tau_{h_{\ell-1}}}{\tau_{h_\ell}} = \frac{Ud^{\frac{1}{h_\ell}}}{n_{h_\ell}}.
\end{equation}

For the case where $h_\ell > 1 + h_{\ell-1}$, the definition of $h_\ell$ indicates that
\begin{equation}
\frac{\tau_{h_{\ell-1}}}{\tau_{h_\ell}} = \frac{(Ud^{\frac{1}{h_\ell}})^{h_{\ell-1} - h_\ell - 1}}{\prod_{j=1}^{h_{\ell-1}} n_j} \leq 2;
\end{equation}
otherwise, $h_\ell$ would not be the smallest integer in $[1 + h_{\ell-1}, d]$ satisfying $\tau_{h_\ell} < \tau_{h_{\ell-1}}/2$. Hence,
\begin{equation}
\frac{\tau_{h_{\ell-1}}}{\tau_{h_\ell}} \leq 2 \cdot \frac{Ud^{\frac{1}{h_{\ell-1}}}}{n_{h_\ell}},
\end{equation}
which completes the proof.