Distributed Streaming with Finite Memory

Frank Neven[1], Nicole Schweikardt[2], Frederic Servais[1], and Tony Tan[1]

1 **Hasselt University and Transnational University of Limburg**
2 **Humbold-University Berlin**

Abstract

We introduce three formal models of distributed systems for query evaluation on massive databases: Distributed Streaming with Register Automata (DSAs), Distributed Streaming with Register Transducers (DSTs), and Distributed Streaming with Register Transducers and Joins (DSTJs). These models are based on the key-value paradigm where the input is transformed into a dataset of key-value pairs, and on each key a local computation is performed on the values associated with that key resulting in another set of key-value pairs. Computation proceeds in a constant number of rounds, where the result of the last round is the input to the next round, and transformation to key-value pairs is required to be generic. The difference between the three models is in the local computation part. In DSAs it is limited to making one pass over its input using a register automaton, while in DSTs it can make two passes: in the first pass it uses a finite-state automaton and in the second it uses a register transducer. The third model DSTJs is an extension of DSTs, where local computations are capable of constructing the Cartesian product of two sets. We obtain the following results: (1) DSAs can evaluate first-order queries over bounded degree databases; (2) DSTs can evaluate semijoin algebra queries over arbitrary databases; (3) DSTJs can evaluate the whole relational algebra over arbitrary databases; (4) DSTJs are strictly stronger than DSTs, which in turn, are strictly stronger than DSAs; (5) within DSAs, DSTs and DSTJs there is a strict hierarchy w.r.t. the number of rounds.

1998 ACM Subject Classification C.2.4 Distributed Systems, H.2.4 Systems, H.2.6 Database Machines

Keywords and phrases Distributed systems, relational algebra, semijoin algebra, register automata, register transducers.

Digital Object Identifier 10.4230/LIPIcs.ICDT.2015.1

1 Introduction

Recent years have seen a massive growth in parallel and distributed computations based on the key-value paradigm. This was fostered by the emergence of popular systems such as Hadoop [33] and Spark [27], which support this paradigm, as well as by many specialised systems built on top of them such as Hive [31], Pig [15], Shark [34], etc.

In brief, the key-value paradigm works as follows. An input dataset D is first transformed into another dataset D' of key-value pairs which is then distributed across a cluster of machines, where values with the same key are sent to the same server. The main computation is performed on D', where values in different servers can be processed in parallel. Take, for example, the Pig Latin[1] script below for computing the query $A(x, y) \wedge \neg B(y)$:

[1] See [26, 25, 15] and the references therein for more details about Pig Latin and the Pig system.

18th International Conference on Database Theory (ICDT'15).
Editors: Marcelo Arenas and Martin Ugarte; pp. 1–42

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
1.  A  =  load 'A.txt' as (x,y);
2.  B  =  load 'B.txt' as (y);
3.  C  =  cogroup A by y, B by y;
4.  D  =  filter C by IsEmpty(B);
5.  E  =  foreach D generate flatten(A);  // E is the result of A(x,y) ∧ ¬B(y)
```

In brief, the Pig system converts this script into a Hadoop's MapReduce program that does the following. The mapper maps each tuple $A(a, b)$ into a key-value pair $(\mathsf{key} = b, \mathsf{val} = A(a, b))$; and each tuple $B(b)$ into $(\mathsf{key} = b, \mathsf{val} = B(b))$. This is done in the script's step 3. For each key c, it checks whether there is an A-tuple and a B-tuple. It collects only those keys in which there is A-tuple, but no B-tuple. This is done in step 4. It will then store only the A-tuples from the collected keys. This is done in step 5.

This example highlights one of the most appealing features of the key-value paradigm: ease of parallelisation. Since computations for different keys are independent, they can be computed in parallel by assigning each key to a server that is responsible for its computation. Typically one server can be assigned with many keys, and in Hadoop such assignments are done using random hash function by default.

We are aware that the key-value paradigm is often called the map-reduce paradigm, and rightly so. However, in many systems communities the name map-reduce refers to Hadoop's MapReduce and excludes Spark, even though Spark does support map-reduce like computations. The difference between map-reduce in Hadoop and Spark lies in, among many other aspects, the implementation of fault tolerance and data storage [27, 35]. Since our focus is on the theory, and to avoid confusion, we opt for the name key-value paradigm.

Many algorithms and systems have been built based on the key-value paradigm. We will discuss some of them in the related work section at the end of Section 1. In the database setting, SQL-queries are *the* standard class of queries. Recently, systems such as Pig, Hive, and Shark have been built to support SQL-like queries on massive datasets, and have been widely used in both academia and industry. However, still lacking is a detailed study of their theoretical foundations.

In this paper we aim to contribute to filling this gap. Our goal is to determine computing mechanisms that are necessary and sufficient for evaluating relational algebra, which is the foundation of the SQL query language. To this end, we introduce three models for distributed computations based on the key-value paradigm and compare their expressiveness with relational algebra: *Distributed Streaming with register Automata* (DSAs), *Distributed Streaming with register Transducers* (DSTs), and *Distributed Streaming with register Transducers and Joins* (DSTJs). In introducing new models, we must be aware that systems like Pig, Hive, or Shark are fully automated in the sense that an input query is automatically converted into a program in Hadoop (for Pig and Hive) or Spark (in the case of Shark). The models must be simple enough to allow for such automation, while still being strong enough to capture a useful class of queries (in our case, relational algebra or suitable fragments thereof).

Brief description of our models. To avoid clutter, we start by defining our models for Boolean queries over directed finite graphs, which is the simplest form of database.

For Boolean queries each model consists of three components: (1) a *mapper* that maps each element in the input to a bag of key-value pairs; (2) a *reducer* that computes for each key separately on the bag of values associated to that key and outputs a bag of values; (3) an *aggregator* that determines the final output yes/no from a bag of values. They can perform multiple rounds of computation, where the output of the reducer is passed as input to the next mapper. The aggregator is consequently only applied at the very end to determine the

result. For non-Boolean queries, we discard the aggregator, and set the output of the last reducer as the output of the computation.[2]

The difference between DSAs/DSTs/DSTJs and the general key-value paradigm lies on the specific, concrete models of computations assigned to the map, reduce, and aggregator functions. In fact, the models assigned are very simple as we will briefly explain below.

In DSAs the mappers are *generic* functions that map *deterministically* a tuple to a bag of key-value pairs based on the equality type of the input tuple. They are essentially functions that neither can invent values nor interpret values, except for the equality test among the data values. The reducers and aggregators in DSAs are *commutative*[3] finite memory automata [17], also called register automata [23]. These are finite automata extended with a fixed number of registers where each register can hold a data value. The automata change states depending on the current state and equality tests among the values currently stored in the registers and those in the input tuple. In the reduce phase, the input values are fed to the automaton one by one (hence, the name "streaming"). After having read the last input item, it outputs a finite bag of values of constant size depending on the final configuration. The automaton from the aggregator component is used to pass through the output of the last reduce phase to determine the end result.

Note that the computation performed by a DSA's mapper, reducer, or aggregator process the input only once while using at most logarithmic space. Furthermore, the number of elements in the output of reducers within the DSA model does not depend on the length of its input but only on the reducer itself. Hence, DSAs are rather limited as they need to summarise an input stream by a fixed number of output values. In particular, DSAs cannot transform a stream of values into another stream of values. To allow for this, we introduce the second model, DSTs. DSTs use the same mappers and aggregators as DSAs, but it has available more powerful reducers. In DSTs, a reducer makes two passes over the input: in the first pass it uses a commutative finite state automaton to gather some finite information on the input, and in the second pass it uses a commutative *register transducer* that for each input value outputs a bag of values. A register transducer works essentially like a register automaton. It has a fixed number of registers where each register can hold a data value. Depending on the current state and the equality tests among the values in the registers and in the input tuple, it can change its state and at the same time output a bag of values. Hence, a register transducer can transform a stream of values into another stream of values.

Note that it seems very unlikely that DSAs or DSTs can compute Boolean queries that involve join operations, where there can be a quadratic blow-up in the size of intermediate results. In fact, as we will show later, both DSAs and DSTs cannot detect the existence of a triangle in a given graph, and hence, cannot perform join operations. This motivates us to introduce the third model called DSTJs. Again, the only difference between DSTJs and DSTs lies on the reducers. In DSTJs, the reducers can be of two types: a register transducer (as used by DSTs), or an abstract function that performs a Cartesian product between two subsets of the input values; the latter is a natural abstraction of the `join` transformation supported by the RDD data structure in Spark [28, 27, 35]. By definition, DSTJs hence can perform join operations. We will show later that DSTJs can evaluate the whole class of

[2] We note that although the aggregator component is not common in the key-value paradigm, it does exist. See, for example, the system Bagel [8]. For Boolean queries such as *"Are there at least 1000 triangles?"*, it is more convenient and efficient to add an aggregator component that aggregates all the output, rather than adding an extra round to simulate the aggregator.

[3] Commutativity is necessary to ensure that the output is independent of the order in which the input tuples are processed.

relational algebra queries.

Main results. The main results in this paper are the following:

(1) DSTJs are strictly stronger than DSTs, which are strictly stronger than DSAs; and within each of the 3 models there is a strict expressiveness hierarchy w.r.t. the number of rounds;
(2) neither DSAs nor DSTs can detect the presence of a triangle in a graph (hence, neither DSAs nor DSTs can do joins);
(3) when restricting attention to bounded degree databases, DSAs can evaluate relational algebra (and, even more, first-order sentences with modulo counting quantifiers)
(4) over arbitrary databases, DSTs can evaluate the semijoin algebra while DSAs can not;
(5) over arbitrary databases, DSTJs can evaluate the relational algebra while DSTs can not.

The relations among DSAs, DSTs and DSTJs with the classical database queries are illustrated as follows.[4]

DSA	$\subsetneq$	DST	$\subsetneq$	DSTJ
$\cup\!\mid$		$\cup\!\mid$		$\cup\!\mid$
FO (= rel. alg.) over bounded degree databases		semijoin algebra		relational algebra

These results emphasise that, albeit simple, DSAs, DSTs, and DSTJs are pretty expressive. In fact, they also highlight that the power of the key-value paradigm here lies within the ability to group values according to a common key.

Related Work. The key-value paradigm, or map-reduce paradigm, attracted a lot of attention since its inception into Google in the mid 2000s [13, 14]. Arguably it can be viewed as a subclass of the BSP model introduced by Valiant back in 1990 [32], in which the keys play a special role in determining the distribution of the data. We discuss the work most related to the setting of the present paper. We are aware of [5, 1, 4, 2, 9, 10, 11, 19, 20, 21, 29, 30]. Karloff et al. [18] introduce a rigorous computation model for the MapReduce programming paradigm where (randomised) mappers and reducers are implemented by a RAM with sublinear space and polynomial time. It is typical that in the map-reduce computation the reducers considered so far in the literature, such as [2, 4, 29], while limited in the number of data it can access, can be arbitrarily strong, typically polynomial time machines in the number of original input items. This is obviously orthogonal with our models here, where the power of the reducers are limited.

Map-reduce as a framework for the evaluation of special classes of queries, especially the join queries, has been considered by a number of articles. However, it is not that clear how to extend them to full relational algebra. We mention here some of the work along this line. Afrati and Ullman [5] study the evaluation of join queries and take the amount of communication, calculated as the sum of the sizes of the input to reducers, as a complexity measure. Evaluation of transitive closure and datalog queries in MapReduce has been investigated in [1, 6]. Afrati et al. [4] study the tradeoff between parallelism and communication cost in a map-reduce setting. In particular, the authors established lower and upper bounds on communication costs for a number of typical problems in databases. All the lower bounds are established only for one round computation.

[4] It is a classic result by Codd [12] that first-order logic and relational algebra are equivalent in terms of expressiveness.

Most of the existing MapReduce algorithms assume the number of keys generated is bounded by a constant, equating the number of keys with the number of available servers. See, for example, the algorithm for enumerating the triangles in [29] and arbitrary sample subgraphs in [2, 4]. This is orthogonal to our approach, where the number of generated keys can be proportional to the number of vertices in the input graph, and parallelisation can be achieved by automatically hashing the keys to the available servers. A more thorough discussion on generic mappers is provided in Section 3.

Koutris and Suciu [19] introduce the massively parallel (MP) model of computation, where computations proceed in a sequence of parallel steps, each followed by a global synchronisation of all servers. In this model, evaluation of conjunctive queries [9, 19] as well as skyline queries [3] have been considered. The MP model can be implemented in the map-reduce setting, with the hash functions fully specified. Again, the bounds, especially the lower bounds, are established mainly for one round of computation.

Another setting, but orthogonal to the MapReduce framework, is that of declarative networking where distributed computations and networking protocols are modeled and programmed using formalisms based on Datalog [7, 16].

Outline. We give a formal definition of the key-value paradigm in Section 2. In Section 3 we present the notion of generic mappers. Then, in Sections 4–6 we provide the formal definitions of DSAs, DSTs, and DSTJs, respectively, and study their expressiveness. In Section 7 we establish the relations between our DSA/DST/DSTJ models and the classical semijoin algebra and relational algebra. We conclude in Section 8.

2 The key-value paradigm

We start by introducing some notations. Let $\mathbb{N}$ be the set of natural numbers $\{1, 2, \ldots\}$. For $m \in \mathbb{N}$, we let $[m] = \{1, \ldots, m\}$. Let S and T be sets. We write $\mathrm{Pow}(S)$ or 2^S to denote the set of all finite subsets of S, and we write $\mathcal{P}(S, T)$ and $\mathcal{F}(S, T)$ to denote the class of all partial functions and all functions from S to T, respectively. We write $\mathrm{Bags}(S)$ to denote the set of all finite *bags* over S (i.e., all finite multisets built from elements in S). Instead of $B \in \mathrm{Bags}(S)$, we sometimes write $B \sqsubseteq S$. We write $\boldsymbol{\chi}_B$ to denote the characteristic function of the bag B. That is, for every $x \in S$, $\boldsymbol{\chi}_B(x)$ returns the multiplicity of x in B. We say that A is a *subbag* of B, if $\boldsymbol{\chi}_A(x) \leq \boldsymbol{\chi}_B(x)$, for every $x \in S$.

We fix an infinite set $\mathbf{D}$ of *data values*. In this paper, we are mostly concerned with (finite, directed) graphs $G = (V, E)$, where $V \subseteq \mathbf{D}$ and $E \subseteq V \times V$. Such graphs are always presented in the form of a sequence of pairs $(a_1, b_1), \ldots, (a_n, b_n)$ where each pair (a_i, b_i) indicates that there is an edge from vertex a_i to vertex b_i. In view of this, elements of $\mathbf{D}$ will also be called vertices, or nodes. We use the words vertex and node interchangeably, and we write $V(G)$ and $E(G)$ to denote G's set of vertices and edges, respectively. We refer to Section 7 for a generalisation to relations of higher arity, where the input is a stream of facts of the form $R(a_1, \ldots, a_m)$, where R is an arbitrary relation symbol.

We assume we are given two sets $\mathbf{K}$ and $\mathbf{V}$ denoting the domain of *keys* and *values*, respectively. We call an element $(\mathsf{key}, \mathsf{val}) \in \mathbf{K} \times \mathbf{V}$ a *key-value pair* and an element $(\mathsf{key}, B) \in \mathbf{K} \times \mathrm{Bags}(\mathbf{V})$ a *key-bag-value pair*. To differentiate them from the standard tuple, we will write $\langle \mathsf{key} : \mathsf{val} \rangle$ and $\langle \mathsf{key} : B \rangle$ to denote key-value and key-bag-value pairs, respectively.

A *key-value paradigm* (KVP) instance is a tuple $\mathcal{M} = (map_1, red_1, \ldots, map_\ell, red_\ell, agg)$ where $\ell \in \mathbb{N}$. We say that $\mathcal{M}$ has ℓ *rounds*. The components of $\mathcal{M}$ are defined as follows:

- map_1 is an *initial mapper* which maps an edge $e \in \mathbf{D} \times \mathbf{D}$ to a finite bag over $\mathbf{K} \times \mathbf{V}$;
- for each $i \geq 2$, map_i is a *mapper* which maps a value in $\mathbf{V}$ to a finite bag over $\mathbf{K} \times \mathbf{V}$;

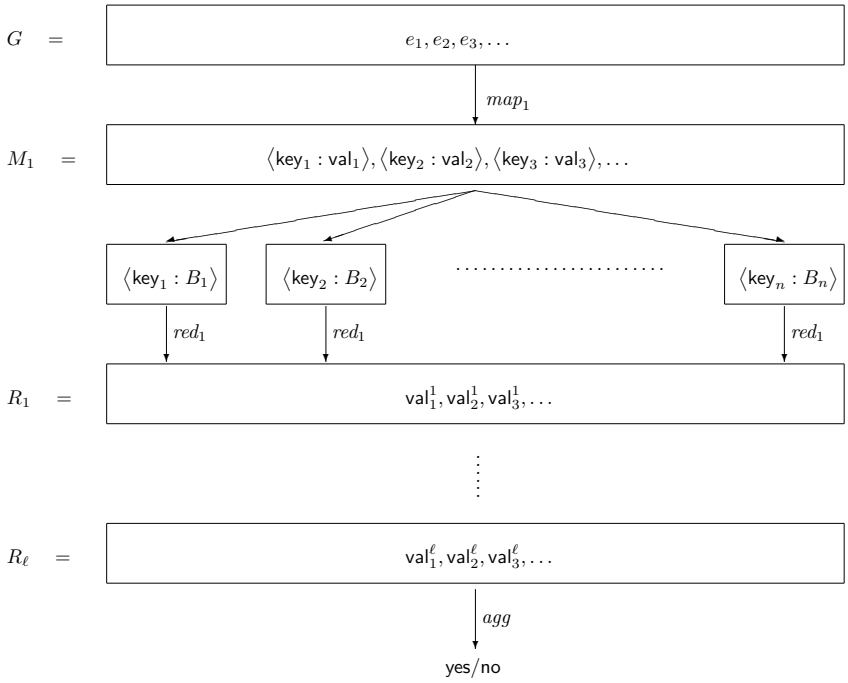


Figure 1 The flow of computation in an ℓ round key-value paradigm computation.

- for each $i \in [\ell]$, red_i is a *reducer* which maps a key $\mathsf{key} \in \mathbf{K}$ and finite bag $B \sqsubseteq \mathbf{V}$ of values to a finite bag $B' \sqsubseteq \mathbf{V}$ of values; and,
- agg is an *aggregator* which determines the output of $\mathcal{M}$; agg is a function mapping a finite bag over $\mathbf{V}$ to the value yes or no.

For a bag $B \sqsubseteq \mathbf{K} \times \mathbf{V}$, define $\text{keys}(B) = \{\mathsf{key} \mid \exists\, \mathsf{val} \in \mathbf{V}, \langle \mathsf{key} : \mathsf{val}\rangle \in B\}$ as the set of keys occurring in B, and $\text{values}(\mathsf{key}, B) = \{\!\{\mathsf{val} \mid \langle \mathsf{key} : \mathsf{val}\rangle \in B\}\!\}$ as the *bag* of values occurring in B with key key. Here, we use double braces $\{\!\{...\}\!\}$ to indicate bags, i.e., if B contains i copies of tuple $\langle \mathsf{key} : \mathsf{val}\rangle$, then $\text{values}(\mathsf{key}, B)$ contains i copies of val.

On input $G = (V, E)$, the output $\mathcal{M}(G) \in \{\mathsf{yes}, \mathsf{no}\}$ is computed as follows:

- $M_1 = \bigcup_{e \in E} map_1(e)$ and $R_1 = \bigcup_{\mathsf{key} \in \text{keys}(M_1)} red_1(\mathsf{key}, \text{values}(\mathsf{key}, M_1))$.
- For each $i \in \{2, \ldots, \ell\}$,

$$M_i = \bigcup_{\mathsf{val} \in R_{i-1}} map_i(\mathsf{val}) \quad \text{and} \quad R_i = \bigcup_{\mathsf{key} \in \text{keys}(M_i)} red_i(\mathsf{key}, \text{values}(\mathsf{key}, M_i))$$

- Finally, $\mathcal{M}(G) = agg(R_\ell)$.

We write $M_i(G)$ and $R_i(G)$ to indicate that the bags M_i and R_i are obtained when the input graph is G. We say that $\mathcal{M}(G)$ is the output of $\mathcal{M}$ on input graph G.

Figure 1 illustrates the flow of computation in an ℓ-round KVP instance. As mentioned in Section 1, the models DSA/DST/DSTJ introduced in this paper follow the key-value paradigm, where the mappers are required to be generic (see Section 3), and the reducers and aggregators are specified by extensions of finite automata (see Sections 4–6).

3 Generic mappers

In this section we instantiate the key and value sets $\mathbf{K}$ and $\mathbf{V}$, and define formally the notion of generic mappers. We fix a finite alphabet Σ and a number $k \in \mathbb{N}$. We reserve $\#$ to be

a special symbol not in $\mathbf{D}$, intended to represent an empty spot or an empty register. $\mathbf{D}_\#$ denotes the set $\mathbf{D} \cup \{\#\}$. We usually write $a, b, c, \ldots$ to denote elements of $\mathbf{D}_\#$ and $\bar{a}, \bar{b}, \bar{c}, \ldots$ for elements of $\mathbf{D}_\#^k$ with $k \in \mathbb{N}$. When $\bar{a} \in \mathbf{D}_\#^k$, we tacitly assume that $\bar{a} = a_1, \ldots, a_k$.

Define $\mathbf{A}_k$ as $\Sigma \times \mathbf{D}_\#^k$. Both $\mathbf{K}$ and $\mathbf{V}$ will be interpreted as $\mathbf{A}_k$. The purpose of σ in $(\sigma, \bar{a}) \in \Sigma \times \mathbf{D}_\#^k$ is to encode a finite amount of information about the vertices in $\bar{a}$. For $t = (\sigma, \bar{a}) \in \mathbf{A}_k$, we call σ the label of t.

A $\mathbf{D}$-*bijection* is a 1-1 mapping $\pi : \mathbf{D}_\# \to \mathbf{D}_\#$, where $\pi(\#) = \#$. We extend π to tuples in the canonical way. Let R and S be finite sets and let f be a function from $R \times \mathbf{D}_\#^m$ to $\text{Bags}(S \times \mathbf{D}_\#^n)$ for some $m, n \in \mathbb{N}$. The function f is *generic* if the following two conditions hold: (1) For all $(r, \bar{c}) \in R \times \mathbf{D}_\#^m$, if $(s, \bar{d}) \in f(r, \bar{c})$, then all non-# values in $\bar{d}$ are from $\bar{c}$; i.e., f cannot invent new values. (2) For every $\mathbf{D}$-bijection π, $\boldsymbol{\chi}_{f(r,\bar{c})}(s, \bar{d}) = \boldsymbol{\chi}_{f(r,\pi(\bar{c}))}(s, \pi(\bar{d}))$; i.e., f cannot interpret values in $\mathbf{D}$.

Let us briefly comment on our choice of generic mappers. In the theoretical studies of MapReduce computations, a mapper is typically a hash function, which maps the data items to the available machines; see, for example, [5, 2, 9, 10, 19, 21, 29]. This is different to our model here, where the mappers are generic functions that map a value deterministically to a set of key-value pairs. Such mappers are not uncommon. For example, the mappers generated by the Pig system [15] are essentially generic mappers similar to the ones studied in this paper; see [25, Section 4.2]. We will give a more detailed comparison between our model and the Pig system at the end of Section 7.

Obviously, the generic mappers can generate as many keys as the number of tuples in the input database. However, this does not mean that the system needs one machine for one key. In the classic example of a MapReduce program for "word count" [13], the mapper is a generic function and the number of keys produced equals the number of different words in the input text. But one would hardly insist that it requires one machine for each key. Rather, to achieve parallelisation, the system automatically hashes the keys to the available machines[5], and the processor evaluates the values for each key separately, one key at a time. Of course, specific hash functions may be desirable to achieve optimisation in some settings, say when the input datasets have been preprocessed, or when some statistics about the input are known. This is out of the scope of our paper. Our goal is to study the sufficient and necessary computation mechanism to evaluate relational algebra in a general setting, where nothing is known about the data or the available machines.

To end this section, let us describe how generic mappers can be specified. We let $[k]_\# := [k] \cup \{\#\}$. The *equality type* τ of a tuple $(d_1, \ldots, d_k)$ is the undirected graph with vertex set $[k]_\#$, where for $i, j \in [k]$ there is an edge between vertices i and j iff $d_i = d_j$, and there is an edge between vertices i and # iff $d_i = \#$. A generic mapper can be specified by a table that assigns to each *equality type* τ over $[k]_\#$ a list $p_1, \ldots, p_s$ of *patterns*, each of the form $\langle k_i : v_i \rangle$, where $k_i = (\sigma_i, j_1, \ldots, j_k)$ and $v_i = (\sigma_i', j_1', \ldots, j_k')$ with $\sigma_i, \sigma_i' \in \Sigma$ and $j_1, \ldots, j_k, j_1', \ldots, j_k' \in [k]_\#$. On input of a tuple $(\sigma, d_1, \ldots, d_k) \in \mathbf{A}_k$, the mapper then determines the equality type τ of $(d_1, \ldots, d_k)$, looks up the according patterns $p_1, \ldots, p_s$, and for each such p_i outputs the key-value pair $\langle \mathsf{key}_i : \mathsf{val}_i \rangle$ with $\mathsf{key}_i = (\sigma_i, d_{j_1}, \ldots, d_{j_k})$ and $\mathsf{val}_i = (\sigma_i', d_{j_1'}, \ldots, d_{j_k'})$, where $d_\#$ is defined to be the value #. Generic *initial* mappers are specified accordingly, where only equality types over $\{1, 2, \#\}$ for input tuples $(d_1, d_2) \in \mathbf{D} \times \mathbf{D}$ are considered.

[5] By default, the Hadoop system [33] takes a random hash function to hash the keys, which in practice works well. Theoretically this is not surprising. A standard application of Chernoff bounds guarantees that the keys are assigned to all machines uniformly (up to a small constant factor). Nevertheless, Hadoop also provides a platform for the user to specify his/her own hash functions.

4 Distributed streaming with register automata (DSA)

In this section we introduce DSAs and study their expressiveness. We start with RA-reducers and RA-aggregators, which are reducers and aggregators instantiated with register automata. Following this, we present the formal definition of DSAs, and establish their expressiveness, as well as a hierarchy on the number of rounds.

RA-reducers. We start with the notion of register transition systems, which are essentially register automata [17, 23]. Intuitively, they work as follows. The input is a sequence of elements of $\mathbf{A}_k$, and each register can hold an element of $\mathbf{D}_\#$. For every input $(\sigma, \bar{a}) \in \mathbf{A}_k$, the system changes its state depending on σ and equality tests among the vertices in $\bar{a}$ and the vertices currently stored in the registers. The formal definition reads as follows.

▶ **Definition 1.** For $r \in \mathbb{N}$, an *r-register transition system over* $\mathbf{A}_k$ is a tuple $\mathcal{S} = \langle Q, \delta \rangle$, where $r \geq k$, Q is a finite set of states, and δ is a transition function from $Q \times \Sigma \times \mathcal{F}([k], 2^{[r]})$ to $\mathcal{P}([r], [k]) \times Q$.[6]

The intuitive meaning of a transition in δ is as follows. If on input $(\sigma, \bar{a})$ the system is in state q, and the data value a_i appears in exactly the registers in $f(i)$ for each $i \in [k]$, and $\delta(q, \sigma, f) = (g, q')$, then the system can enter state q' and replace the content of each register j with $a_{g(j)}$ for each $j \in [r]$.

A *configuration of* $\mathcal{S}$ is an element of $Q \times \mathbf{D}_\#^r$. An element $(\sigma, \bar{a}) \in \mathbf{A}_k$ induces a relation $\vdash_{(\sigma,\bar{a})}$ on the configurations of $\mathcal{S}$ defined as follows: $(q, \bar{u}) \vdash_{(\sigma,\bar{a})} (q', \bar{v})$, if $\delta(q, \sigma, f) = (g, q')$ and

- $f(i) = \{j \mid u_j = a_i\}$ for each $i \in [k]$, and
- for each $i \in [r]$, if $g(i)$ is defined, then $v_i = a_{g(i)}$ and if $g(i)$ is undefined, then $v_i = u_i$.

Let $t = t_1 \cdots t_n$ be a sequence of elements of $\mathbf{A}_k$. A *run of $\mathcal{S}$ on t starting from a configuration* $(q, \bar{u})$ is a sequence $(q_0, \bar{u}_0), \ldots, (q_n, \bar{u}_n)$ of configurations, where $(q_0, \bar{u}_0) = (q, \bar{u})$ and $(q_{i-1}, \bar{u}_{i-1}) \vdash_{t_i} (q_i, \bar{u}_i)$ for each $i \in [n]$.

We now define reducers in terms of transition systems.

▶ **Definition 2.** An *RA-reducer over* $\mathbf{A}_k$ is a tuple $red = (\mathcal{S}, \rho_{in}, \rho_{out})$, where $\mathcal{S} = \langle Q, \delta \rangle$ is an r-register transition system over $\mathbf{A}_k$ and $r \geq k$; ρ_{in} is a function that maps an element of $\mathbf{A}_k$ to a configuration of $\mathcal{S}$; and, ρ_{out} is a function that maps a configuration of $\mathcal{S}$ to a finite bag over $\mathbf{A}_k$. Both ρ_{in} and ρ_{out} are required to be generic.

Intuitively, each reducer gets as input a key-bag-value pair $\langle \mathsf{key} : B \rangle$ where $\rho_{in}(\mathsf{key})$ identifies the initial configuration from which the run of $\mathcal{S}$ is started. The output then is $\rho_{out}(c)$, where c is the last configuration of the run.

Formally, let $\langle \mathsf{key} : B \rangle \in \mathbf{A}_k \times \mathrm{Bags}(\mathbf{A}_k)$ be a key-bag-value pair, and let $t_1, \ldots, t_m$ be an enumeration of the elements in B.[7] The output $red(\mathsf{key}, B)$ is defined as $\rho_{out}(q_m, \bar{u}_m)$ for the run $(q_0, \bar{u}_0), \ldots, (q_m, \bar{u}_m)$ of $\mathcal{S}$ on $t_1 t_2 \cdots t_m$ with $(q_0, \bar{u}_0) = \rho_{in}(\mathsf{key})$.

Obviously, the run of $\mathcal{S}$ on B depends on the order in which $t_1, \ldots, t_m$ are presented. However, we want to insist that the output $red(\mathsf{key}, B)$ is the same regardless of the order in which the elements in B are arranged. Therefore, we require RA-reducers to be *commutative*

[6] Note that unlike the definition of register automata in [17] and [23], in a transition system we do not specify the initial state, the final states and the initial content of the registers. We will, however, use the standard register automata to define the aggregator.

[7] Since B is a bag, some elements can appear multiple times in the enumeration.

in the following sense: If $t = t_1 \cdots t_m$ and $t' = t_{\pi(1)} \cdots t_{\pi(m)}$ are two enumerations of the elements of B (for some permutation π of $[m]$), and $(q_m, \bar{u}_m)$ and $(q'_m, \bar{u}'_m)$ are the final configurations of the runs of $\mathcal{S}$ on t and t', respectively, starting in configuration $\rho_{in}(\mathsf{key})$, then $\rho_{out}(q_m, \bar{u}_m) = \rho_{out}(q'_m, \bar{u}'_m)$.

Note that by definition of a transition system, an RA-reducer can never get stuck and always processes the complete input. The output of an RA-reducer is therefore well-defined.

RA-aggregator. An *r-register automaton over* $\mathbf{A}_k$ is an r-register transition system $\mathcal{S} = \langle Q, \delta \rangle$ together with a designated initial state q_0, a set of final states $F \subseteq Q$ and an initial content of the registers $\bar{u}_0$. We will write $\mathcal{A} = \langle Q, \delta, q_0, F, \bar{u}_0 \rangle$ to denote an r-register automaton.

The configurations of $\mathcal{A}$ and the relations $\vdash_{(\sigma,\bar{a})}$ are defined similarly as for a transition system. The only difference is that in a register automaton, we insist that the run should start from the configuration $(q_0, \bar{u}_0)$.

Formally, let $t = t_1 \cdots t_n$ be a sequence of elements of $\mathbf{A}_k$. The run $(q_0, \bar{u}_0), \cdots, (q_n, \bar{u}_n)$ of $\mathcal{A}$ on t is *accepting* (and $\mathcal{A}$ *accepts* t) iff $q_n \in F$. The automaton is *commutative* when $\mathcal{A}$ accepts $t_1 \cdots t_n$ if and only if $\mathcal{A}$ accepts $t_{\pi(1)} \cdots t_{\pi(n)}$ for every sequence $t = t_1 \cdots t_n$ of elements of $\mathbf{A}_k$ and for every permutation π on $[n]$. For commutative register automata we can safely regard the input sequence as a finite bag B, where we consider an arbitrary enumeration of the elements in B and in which context we simply say that either $\mathcal{A}$ accepts B or not.

▶ **Definition 3.** An *RA-aggregator* is a commutative r-register automaton $\mathcal{A}$ over $\mathbf{A}_k$ with $r \geq k$.

Obviously, an *RA-aggregator* $\mathcal{A}$ can be viewed as a function from finite bags of $\mathbf{A}_k$ to $\{\mathsf{yes}, \mathsf{no}\}$, where $\mathcal{A}(B) = \mathsf{yes}$, if $\mathcal{A}$ accepts B, and $\mathcal{A}(B) = \mathsf{no}$, otherwise.

Definition of DSA. An ℓ-round DSA is a tuple $\mathcal{M} = (map_1, red_1, \ldots, map_\ell, red_\ell, agg)$, where each map_i is a generic mapper, each red_i is an RA-reducer, and agg is an RA-aggregator.

We say that $\mathcal{M}$ *accepts* a graph G, if $agg(R_\ell(G)) = \mathsf{yes}$, in which case, we write $\mathcal{M}(G) = \mathsf{yes}$. Here, R_ℓ is as defined in Section 2. By $\mathcal{G}(\mathcal{M})$ we denote the class of all graphs accepted by $\mathcal{M}$, and we say that $\mathcal{G}(\mathcal{M})$ is the class of graphs *recognised* by $\mathcal{M}$.

▶ **Example 4.** Consider inputs of the form $(d_1, s_1), \ldots, (d_n, s_n)$, where each tuple (d_i, s_i) indicates that data value d_i is stored on server s_i. Let INTERSECT be the problem to decide whether there is a data value that is stored on more than one server. It can easily be formalised as a 1-round DSA $\mathcal{M} = (map_1, red_1, agg)$ over $\mathbf{A}_k$ for $k = 1$ and $\Sigma = \{\sigma_{\text{blank}}, \sigma_{\text{disj}}, \sigma_{\text{ndisj}}\}$.

The initial mapper map_1 assigns to each input tuple $(d_i, s_i) \in \mathbf{D} \times \mathbf{D}$ a single key-value pair $\langle \mathsf{key} : \mathsf{val} \rangle$ with $\mathsf{key} = (\sigma_{\text{blank}}, d_i)$ and $\mathsf{val} = (\sigma_{\text{blank}}, s_i)$. Thus, the initial mapper can be specified by a table which assigns to each equality type τ the single pattern $p = \langle k : v \rangle$ with $k = (\sigma_{\text{blank}}, 1)$ and $v = (\sigma_{\text{blank}}, 2)$.

The reducer red_1 is an RA-reducer over $\mathbf{A}_k$ (for $k = 1$), with a single register, with state set $Q = \{q_0, q_1, q_2\}$, and with $\rho_{in}(\mathsf{key}) = (q_0, \#)$ for all $\mathsf{key} \in \mathbf{A}_k$. The transition function δ ensures that when reading a symbol $(\sigma, s) \in \mathbf{A}_k$, the RA-reducer proceeds as follows: If the current state is q_0 (i.e., the automaton performs its first step), then the automaton stores the value s in its register and changes to state q_1. If the current state is q_1, and the value s is different from the value stored in the register, then the automaton changes to state q_2;

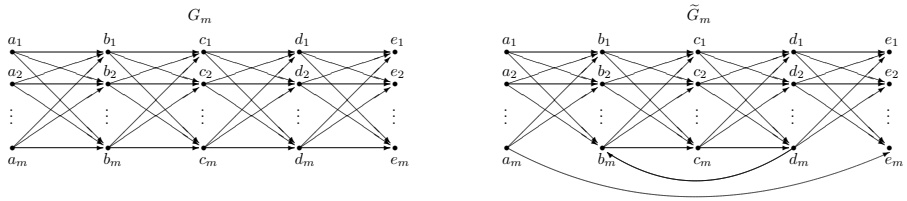


Figure 2 DSAs cannot differentiate between G_m on the left and $\widetilde{G}_m$ on the right.

otherwise (i.e., s coincides with the value stored in the register), the automaton remains in state q_1. If the current state is q_2, then the automaton simply remains in this state.

The function ρ_{out} maps the final configuration (q, v) to $(\sigma_{\text{ndisj}}, \#)$ if $q = q_2$, and to $(\sigma_{\text{disj}}, \#)$ otherwise. Finally, the aggregator *agg* is a simple finite automaton which receives as input a list of items in $\mathbf{A}_k$ and accepts if, and only if, at least one these items is of the form $(\sigma_{\text{ndisj}}, \#)$. This completes the description of a 1-round DSA which solves the INTERSECT problem. □

Expressiveness of DSAs and a hierarchy on the number of rounds. The rest of this section is devoted to our study of the expressiveness of DSAs.

We start by showing that on general graphs DSAs cannot compute joins; in fact, they cannot even test if an input graph contains a triangle. Let TRIANGLE be the class of all graphs G that contain a directed triangle.

▶ **Theorem 5.** *There is no DSA that recognises* TRIANGLE.

Proof (sketch). Consider the graphs G_m and $\widetilde{G}_m$ depicted in Figure 2. While $\widetilde{G}_m$ contains a triangle, G_m does not. We show that for every DSA $\mathcal{M}$ there is an $m \in \mathbb{N}$ such that $\mathcal{M}$ cannot distinguish between G_m and $\widetilde{G}_m$, i.e., $\mathcal{M}(G_m) = \mathcal{M}(\widetilde{G}_m)$. The number m we choose here is bigger than the number r of registers of $\mathcal{M}$, and the proof relies on a careful analysis of the computation of $\mathcal{M}$, utilising the fact that mappers of $\mathcal{M}$ are generic and reducers of $\mathcal{M}$ are generic and commutative. Briefly, it is based on the fact that for every vertex u, its neighbourhoods in both G_m and $\tilde{G}_m$ are "the same". Moreover, since $m \geq r + 1$, by just looking at the u and its neighbourhood, the DSA $\mathcal{M}$ cannot differentiate whether u is a vertex in G_m or $\tilde{G}_m$. This holds for every vertex u in G_m and $\tilde{G}_m$ (both have the same set of vertices), and implies that $\mathcal{M}$ cannot differentiate G_m and $\tilde{G}_m$. ◀

Concerning the graphs G_m and $\widetilde{G}_m$ used in the above proof, note that the maximum length of a walk[8] in G_m is 4, while $\widetilde{G}_m$ contains walks of arbitrary lengths. Thus, we obtain the following where, for $\ell \in \mathbb{N}$, we define ℓ-WALK as the class of all graphs that contain a walk of length ℓ.

▶ **Corollary 6.** *Let $\ell \geq 5$. There is no DSA that recognises ℓ-WALK.*

However, when restricting attention to bounded degree graphs, DSAs are quite powerful: they can recognise all properties definable in first-order logic with modulo counting quantifiers. That is, first-order logic enriched by quantifiers of the form $\exists^{i \bmod m} x\, \psi$, stating that the number of nodes x satisfying ψ is congruent i modulo m, for integers $m \geq 1$ and $i \in \{0, \ldots, m-1\}$.

[8] A walk of length ℓ is a sequence of ℓ edges (a_0, a_1), (a_1, a_2), $\ldots$, $(a_{\ell-1}, a_\ell)$ in which repetition of vertices/edges is allowed.

For a vertex u in a graph G, define in-deg(u) and out-deg(u) as the in-degree and the out-degree of u, respectively, and let deg(u) = in-deg(u) + out-deg(u), and let deg(G) = $\max_{u \in V(G)}(\deg(u))$ be the degree of G.

▶ **Theorem 7.** *Let $d \geq 2$ and let φ be a sentence of first-order logic with modulo counting quantifiers. There is a DSA $\mathcal{M}_{\varphi,d}$ such that $\mathcal{G}(\mathcal{M}_{\varphi,d}) = \{G : \deg(G) \leq d \text{ and } G \models \varphi\}$.*

For $d, \ell \geq 0$, define 2^ℓ-WALK$_d$ to be the class of all graphs G such that deg(G) $\leq d$ and there is a walk of length 2^ℓ in G.

▶ **Theorem 8.** *(1) For every $d, \ell \geq 0$, there is an ℓ-round DSA $\mathcal{M}$ such that $\mathcal{G}(\mathcal{M}) = (2^\ell)$-WALK$_d$.*
(2) For every $\ell \geq 0$, there is no ℓ-round DSA that recognises $(2^{\ell+1})$-WALK$_2$.
(3) For every $\ell \in \mathbb{N}$, $(\ell{+}1)$-round DSAs are strictly more expressive than ℓ-round DSAs.

5 Distributed streaming with register transducers (DST)

In this section we introduce the model DST, which is stronger than the DSA-model. As mentioned earlier, the only difference between DSTs and DSAs is on the reducer level. Within a DSA, a reducer is a register automaton that makes one pass over its input, and upon finishing this pass, it outputs a finite bag of values determined by its final configuration. In contrast, within a DST, a reducer is an *RT-reducer* which consists of two components: a finite-state automaton and a transducer system; and makes *two* passes over the input. In the first pass, it uses its finite-state automaton to read the input, but does not produce any output. The final state of the first pass serves as the initial state for the transducer system to make another pass on the input. During this second pass, the transducer outputs a bag of values for each input value (hence the name transducer).

In the next few paragraphs we present the formal definition of DSTs. We start by extending Definition 1 to transducer systems.

▶ **Definition 9.** For $r \in \mathbb{N}$, an *r-register transducer system over* $\mathbf{A}_k$ is a tuple $\mathcal{T} = \langle Q, \delta, \mu \rangle$, where $r \geq k$, Q is a finite set of states, δ is a transition function from $Q \times \Sigma \times \mathcal{F}([k], 2^{[r]})$ to $\mathcal{P}([r], [k]) \times Q$, and μ is a transducer function from $Q \times \Sigma \times \mathcal{F}([k], 2^{[r]})$ to Bags($\Sigma \times \mathcal{F}([k], [r{+}k])$).

Thus, an r-register transducer system $\mathcal{T} = \langle Q, \delta, \mu \rangle$ is a transition system $\langle Q, \delta \rangle$ extended with a transducer function μ. The meaning of δ is the same as before, while the meaning of μ is as follows. If on input $(\sigma, \bar{a})$ the automaton is in configuration $(q, \bar{u})$, and for each $i \in [k]$, the data value a_i appears in exactly the registers in $f(i)$, then the transducer function outputs the finite bag $C \sqsubseteq \mathbf{A}_k$ which is obtained from $\tilde{C} := \mu(q, \sigma, f)$ by replacing every $(\sigma', h) \in \tilde{C}$ with the value $(\sigma', \bar{v})$ where, for each $i \in [k]$,

$$v_i \quad = \quad \begin{cases} u_{h(i)} & \text{if } h(i) \leq r \\ a_{h(i)-r} & \text{if } h(i) \geq r{+}1 \end{cases}$$

(i.e., the function h tells us for each of the k positions i of $\bar{v}$, that the value at this position should be the value at the $h(i)$-th position of the tuple $\bar{u}\bar{a}$). We say that C is the output of μ from $(\sigma, \bar{a})$ and $(q, \bar{u})$.

Let $t = t_1 \cdots t_n$ be a sequence of elements of $\mathbf{A}_k$. When starting with a configuration $(q, \bar{u})$, the transducer system $\mathcal{T} = \langle Q, \delta, \mu \rangle$ processes t as follows: It runs the transition system $\langle Q, \delta \rangle$ on t starting with configuration $(p_0, \bar{v}_0) := (q, \bar{u})$, resulting in a run

$(p_0, \bar{v}_0), \ldots, (p_n, \bar{v}_n)$. During this run, on reading each t_i it outputs the bag C_i, defined as the output of μ from t_i and $(p_{i-1}, \bar{v}_{i-1})$.

The union C of the bags $C_1, \ldots, C_n$ is the output of the transducer system $\mathcal{T}$ on t from the configuration $(q, \bar{u})$.[9]

▶ **Definition 10.** An *RT-reducer over* $\mathbf{A}_k$ is a tuple $red = (\mathcal{A}, \mathcal{T}, \rho_{in})$, where $\mathcal{A}$ is a commutative finite-state automaton[10] over the alphabet Σ and $\mathcal{T}$ is an r-register transducer system over $\mathbf{A}_k$ for $r \geq k$, and ρ_{in} is a function that maps an element of $\mathbf{A}_k$ to a state of $\mathcal{A}$. As before, ρ_{in} is required to be generic.

As input, an RT-reducer $red = (\mathcal{A}, \mathcal{T}, \rho_{in})$ receives a key-bag-value pair $\langle \mathsf{key} : B \rangle \in \mathbf{A}_k \times \mathrm{Bags}(\mathbf{A}_k)$. Let $t = t_1 \cdots t_m$ be an enumeration of the elements in B. First, the finite state automaton $\mathcal{A}$ reads only the labels in t starting from the state $\rho_{in}(\mathsf{key})$, and ends in a configuration, say q. Then, the transducer system $\mathcal{T}$ reads t starting from the configuration $(q, \bar{a})$, where $\bar{a}$ is the data values component in key. The output of *red* on $\langle \mathsf{key} : B \rangle$ is the output of $\mathcal{T}$ on t. As in the case of RA-reducers, we want to insist that the output $red(\mathsf{key}, B)$ is independent of the order of elements in B read by $\mathcal{T}$. Therefore, we require RT-reducers to be *commutative*.

Finally, we are ready to define DST.

▶ **Definition 11.** An ℓ-round DST is a tuple $\mathcal{M} = (map_1, red_1, \ldots, map_\ell, red_\ell, agg)$, where each map_i is a generic mapper, each red_i is an RT-reducer, and agg is an RA-aggregator.

The notion of acceptance, along with the notions $\mathcal{M}(G)$ (for a graph G) and $\mathcal{G}(\mathcal{M})$, are defined in the same way as for DSAs. Note that we require the reducer to make two passes on the values, where a finite state automaton is making the first pass, and a transducer is making the second pass. Without two passes, semijoin algebra cannot be captured. The rest of this section is devoted to our study of the expressiveness of DSTs.

Our first result states that for DSTs, ℓ rounds are sufficient and necessary to recognise the existence of a walk of length 2ℓ. Recall that ℓ-WALK (for $\ell \in \mathbb{N}$) is the class of all graphs that contain a walk of length ℓ.

▶ **Theorem 12.** *(1) For each $\ell \in \mathbb{N}$ there is an ℓ-round DST $\mathcal{M}$ such that $\mathcal{G}(\mathcal{M}) = (2\ell)$-WALK.*
(2) For each $\ell \in \mathbb{N}$, there is no ℓ-round DST that recognises $(2\ell{+}2)$-WALK.
(3) For every $\ell \in \mathbb{N}$, $(\ell{+}1)$-round DSTs are strictly more expressive than ℓ-round DSTs.

In particular, 6-WALK can be recognised by a DST. From Corollary 6, we know that no DSA can recognise 6-WALK. Furthermore, by modifying the proof of Theorem 5, we can also show that DSTs are still not powerful enough to solve the TRIANGLE problem. These two facts are stated formally as follows:

▶ **Theorem 13.** *DSTs are strictly stronger than DSAs.*
There is no DST that recognises TRIANGLE.

[9] We should remark that although register transducers are very natural extension of register automata, we are not aware of any literature where they have been studied previously.
[10] A finite state automaton $\mathcal{A}$ is commutative, if for every sequence $\sigma_1 \cdots \sigma_m$, for every permutation π on $[m]$, on reading the sequence $\sigma_1 \cdots \sigma_m$ and $\sigma_{\pi(1)} \cdots \sigma_{\pi(m)}$, the automaton ends in the same state.

6 Distributed streaming with register transducers and joins

In this section we introduce the strongest model of this paper, called *Distributed streaming with register transducers and joins* (DSTJ). It is designed specifically to capture relational algebra. The difference between DSTJs and DSTs is again on the reducer level. In DSTJs, a reducer can be of two types: an RT-reducer or a joiner, which is simply an abstract function that performs the Cartesian product between two sets. Its formal definition is as follows.

A *joiner* is a triplet $\mathcal{J} = (\alpha, \beta, \gamma)$, where α, β, γ are symbols from Σ. A joiner $\mathcal{J} = (\alpha, \beta, \gamma)$ works as follows. The input is a key-bag-value pair $\langle \mathsf{key} : \mathsf{VAL} \rangle$. Let $\mathsf{key} = (\zeta, \bar{a})$. The joiner $\mathcal{J}$ outputs the bag $\{\!\{(\alpha, \bar{a}\bar{b}\bar{c}) \mid (\beta, \bar{b}) \in \mathsf{VAL} \text{ and } (\gamma, \bar{c}) \in \mathsf{VAL}\}\!\}$.

Next, we define a *relational reducer* as a reducer that can choose either an RT-reducer or a joiner to process its values.

▶ **Definition 14.** A *relational reducer* is a tuple $\mathcal{R} = (F, \mathcal{J}, \mathcal{T})$, where $F : \Sigma \to \{C, T\}$ is a function that maps $\sigma \in \Sigma$ to either C or T, $\mathcal{J}$ is a joiner, and $\mathcal{T}$ is an RT-reducer.

On input of a key-bag-value pair $\langle \mathsf{key} : \mathsf{VAL} \rangle$, a relational reducer does the following: Let $\mathsf{key} = (\sigma, t)$. If $F(\sigma) = C$, the relational reducer runs the joiner $\mathcal{J}$ on $\langle \mathsf{key} : \mathsf{VAL} \rangle$. If $F(\sigma) = T$, it runs the RT-reducer $\mathcal{T}$ on $\langle \mathsf{key} : \mathsf{VAL} \rangle$.

▶ **Definition 15.** An ℓ-round DSTJ is a tuple $\mathcal{M} = (map_1, red_1, \ldots, map_\ell, red_\ell, agg)$, where each map_i is a generic mapper, each red_i is a relational reducer, and agg is an RA-aggregator.

The notion of acceptance, along with the notions $\mathcal{M}(G)$ (for a graph G) and $\mathcal{G}(\mathcal{M})$, are defined in the same way as for DSTs. The rest of this section is devoted to the expressiveness of DSTJs. Our first expressiveness result states that DSTJ can recognise the existence of a triangle.

▶ **Lemma 16.** *There is a 2-round DSTJ $\mathcal{M}$ such that $\mathcal{G}(\mathcal{M}) =$ TRIANGLE.*

Proof (sketch). Intuitively, in the first round $\mathcal{M}$ collects all pairs (u, v) where there is path of length 2 from u to v. In the second round on each pair (u, v) output in the first round, it checks whether there is an edge from v to u. If so, it outputs a special symbol γ. The aggregator simply checks whether γ appears among the values output by the reducer in the second round. We note that this algorithm is very similar to the algorithm MR-Node-Iterator++ in [29]. ◀

Combining Lemma 16 and Theorem 13, we obtain:

▶ **Theorem 17.** *DSTJs are strictly stronger than DSTs.*

In a graph G, a cycle of length m is sequence of edges $(u_1, u_2), \ldots, (u_{m-1}, u_m), (u_m, u_1) \in E(G)$. It is not necessary that the vertices $u_1, \ldots, u_m$ are pairwise different. For $m \geq 3$, define the class m-CYCLE where a graph $G \in m$-CYCLE if and only if G contains a cycle of length m.

▶ **Theorem 18.** *For each positive integer $\ell \geq 1$, the following holds.*

(1) There is an ℓ-round DSTJ $\mathcal{M}$ such that $\mathcal{G}(\mathcal{M}) = 2^\ell$-CYCLE.
(2) For each $\ell \in \mathbb{N}$, there is no ℓ-round DSTJ $\mathcal{M}$ such that $\mathcal{G}(\mathcal{M}) = 2^{\ell+1}$-CYCLE.
(3) $(\ell{+}1)$-round DSTJs are strictly more expressive than ℓ-round DSTJs.

7 Semijoin algebra and relational algebra

In this section we study the connections between the models DSA/DST/DSTJ and the semijoin algebra and the relational algebra. To this end, we define the corresponding model for DSA/DST/DSTJ for non-Boolean queries on general databases.

We fix a finite vocabulary τ of relation symbols with associated arities and assume every database DB to be over τ. For a relation symbol R and a tuple of values $\bar{a}$ whose arity matches the arity of R, we call $R(\bar{a})$ a *fact*. Clearly, a database is just a finite set of facts. The initial mapper will now receive as input an enumeration of all the facts in the database.

Here we assume that Σ contains τ, and as before, $\mathbf{A}_k$ denotes $\Sigma \times \mathbf{D}^k_\#$. A fact $R(\bar{a})$ can then be viewed as an element of $\mathbf{A}_k$ by padding an appropriate number of #'s at the end of $\bar{a}$. Similarly, an element $(R, \bar{a}) \in \mathbf{D}^k_\#$ can be viewed as an R-fact by discarding the # components. To avoid being pedantic, we will view elements of $\mathbf{A}_k$ as facts, and vice versa.

▶ **Definition 19.** For every $X \in \{\text{DSA}, \text{DST}, \text{DSTJ}\}$, an ℓ-round *DB-X* over $\mathbf{A}_k$ is a tuple $\mathcal{M} = (map_1, red_1, \ldots, map_\ell, red_\ell)$, where each map_i is a generic mapper, and each red_i is an RA-reducer, an RT-reducer, and relational reducer, when X is DSA, DST and DSTJ, respectively.

On an input database DB, for each $i \in [\ell]$, the bags $M_i(\mathsf{DB})$ of key-value pairs and the bags $R_i(\mathsf{DB})$ of values are defined as in Section 2. For every $X \in \{\text{DSA,DST,DSTJ}\}$, on input of a database DB, the output of a DB-X $\mathcal{M}$ is defined as the tuples from $R_\ell(\mathsf{DB})$.

Note that in the lower bounds proved in the previous sections are for models with aggregator components, which the non-Boolean models do not have. Obviously, all the lower bounds for the Boolean queries carry over to their non-Boolean counterparts. Next, we are going to show that on classes of bounded degree databases, DB-DSA can evaluate the relational algebra; while over general databases, DB-DST and DB-DSTJ can evaluate the semijoin algebra and the relational algebra, respectively. In the following $e(\mathsf{DB})$ denotes the result of evaluating the expression e on the database DB.

▶ **Theorem 20.** *(1) For every relational algebra expression e and an integer $d > 0$, there is a DB-DSA $\mathcal{M}_e$ such that for every database DB of degree at most d, $\mathcal{M}_e(\mathsf{DB}) = e(\mathsf{DB})$.*

(2) For every semijoin algebra expression e, there is a DB-DST $\mathcal{M}_e$ such that for every database DB, $\mathcal{M}_e(\mathsf{DB}) = e(\mathsf{DB})$.

(3) For every relational algebra expression e, there is a DB-DSTJ $\mathcal{M}_e$ such that for every database DB, $\mathcal{M}_e(\mathsf{DB}) = e(\mathsf{DB})$.

Moreover, each $\mathcal{M}_e$ can be constructed effectively.

Proof. Proof of (1). We are going to show that on bounded degree databases, each RA operation can be simulated by one round DSA $\mathcal{M} = (map, red)$, in which the tuples output by the reducer has the same label T. Its generalisation for arbitrary RA-expression can be established via straightforward induction. Note that the bounded degree is only needed for the semijoin and join operations.

- Union: $R \cup S$.
  The mapper works as follows. On input t, if t is $R(\bar{a})$, it outputs $\langle T(\bar{a}) : R(\bar{a})\rangle$; if t is $S(\bar{a})$, it outputs $\langle T(\bar{a}) : S(\bar{a})\rangle$; otherwise, it outputs nothing. The reducer *red* works as follows. On key t, it outputs t itself.
- Intersection: $R \cup S$.
  The mapper works like in the case $R \cup S$. The reducer *red* works as follows. On key t, it checks whether there are two tuples, one with label R and another with label S. If so, it outputs t itself. Otherwise, it outputs nothing.

- Difference: $R - S$.
  The mapper works like in the case $R \cup S$. The reducer *red* works as follows. On key t, it checks whether there is a tuple with label R and there is no tuple with label S. If so, it outputs t itself. Otherwise, it outputs nothing.
- Selection: $\sigma_{i=j}(R)$.
  The mapper works as follows. On input t, if t is $R(\bar{a})$ and $a_i = a_j$ it outputs $\langle T(\bar{a}) : T(\bar{a})\rangle$; otherwise, it outputs nothing. The reducer *red* works as follows. On key t, it outputs t itself.
- Projection: $\pi_{i_1,\ldots,i_m}(R)$.
  The mapper works as follows. On input t, it outputs $\langle T(a_{i_1},\ldots,a_{i_m}) : T(a_{i_1},\ldots,a_{i_m})\rangle$, if t is $R(\bar{a})$. Otherwise, it outputs nothing. The reducer *red* works as follows. On key t, it outputs t itself.
- Semijoin: $R \ltimes_\theta S$.
  Let I and J be the projection of θ to its first and second coordinates. The mapper works as follows. On input t, if t is $R(\bar{a})$, it outputs $\langle T(\pi_I(\bar{a})) : R(\bar{a}))\rangle$; if t is $S(\bar{a})$, it outputs $\langle T(\pi_J(\bar{a})) : S(\pi_J(\bar{a})))\rangle$; otherwise, it outputs nothing. The reducer *red* works as follows. On key t, it passes through its input, while remembering all the R-facts in its registers. Since the input dataabse DB is of bounded degree, say $\leq d$, the number R-facts associated with one particular key is also bounded by d. So we can choose the number of registers in *red* to be kd, where k is the arity of R, to accommodate all the R-facts and S-facts. If there is at least an S-fact among its input, it outputs all the R-facts. Otherwise, if there is no S-fact among its input, it outputs nothing.
- Join: $R \bowtie_\theta S$.
  As in the semijoin case, let I and J be the projection of θ to its first and second coordinates. The mapper works in the same manner as in the semijoin case. The reducer *red* works as follows. On key t, it passes through its input, while remembering all the R-facts and all its S-facts in its registers. Similar to the semijoin case, since DB is of bounded degree, say $\leq d$, the number R-facts and S-facts associated with one particular key is also bounded by d. So we can choose the number of registers in *red* to be $(k+l)d$, where k and l are the arities of R and S, respectively, to accommodate all the R-facts and S-facts. If there is at least one R-fact and one S-fact among its input, it outputs all the combination of the join among the R-facts and S-facts in the input. Otherwise, if there is no S-fact or if there is no R-fact, it outputs nothing.

Proof of (2). Note that for union, intersection, difference, selection and projection, one-round DSA presented above works for arbitrary database. Hence, it is sufficient to show that semijoin operation $R \ltimes_\theta S$ over arbitrary graph can be done in one-round DST.

Let I and J be the projection of θ to its first and second coordinates. The mapper works as in the case of semijoin above. The reducer *red* works as follows. On key t, in the first pass it checks whether there is an S-fact among the input, which can be done trivially by a finite state automaton. If there is an S-fact, in the second pass on each R-fact $R(\bar{a})$ in the input, it outputs $T(\bar{a})$. If there is no S-fact, in the second pass it does nothing and output nothing.

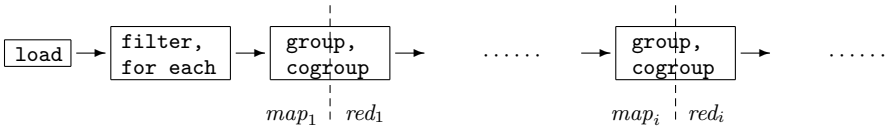
Proof of (3): Again, since all the other operations can be evaluated by DSA and DST, it is sufficient to show that join operation $R \bowtie_\theta S$ over arbitrary database can be done in one-round DSTJ. The mapper works similarly as in the case of join above. Then the reducer uses joiner to pair off the R-tuples with S-tuples. ◀

Note that 6-WALK can be expressed in the semijoin algebra, and TRIANGLE can be expressed in the relational algebra. Thus, it follows that DB-DSA and DB-DST cannot

evaluate all semijoin algebra and relational algebra expressions, respectively.

Comparison with Pig. To end this section, we give a brief description of the Pig system, and relate it to our models here. For more details, we refer the reader to [15, 25]. In short, Pig is a system built on the Hadoop system to evaluate queries on a large relational database written in a language called *Pig Latin*. Upon receiving an input query, Pig generates a MapReduce program that evaluates the query on a given database, where the number of rounds corresponds linearly to the number of `(CO)GROUP` and `JOIN` queries. For each `(CO)GROUP` query it generates a mapper that assigns keys to tuples based on the `BY` clauses in the query, i.e. projecting the tuples to fields in the `BY` clauses. The `JOIN` operations are handled in one of two ways: (i) rewrite into a `COGROUP` followed by a `FOR EACH` operation, which yields a parallel hash-join or sort-merge join, or (ii) use fragment-replicate join. Either way requires one round of MapReduce computation, and can be captured by the joiner.

Obviously, two independent subqueries can be evaluated simultaneously in a one round MapReduce job. Typically a MapReduce compilation of Pig Latin script looks as follows:



The `(CO)GROUP` commands form the boundary between the map and reduce phase. In the current implementation of Pig, the commands in between the boundaries are pushed into the reduce function. Obviously, the `FILTER` and `FOR EACH` command can be implemented as one of RA-reducer or RT-reducer, and `JOIN` as joiner. Hence, one round in the Pig system corresponds to one round of either DSA, DST, or DSTJ.

8 Conclusion

We introduced three simple abstractions of the key-value paradigm in terms of finite memory automata and transducers. Our results emphasise that, even though the proposed models are simple, they form a relevant subclass of MapReduce. In particular, DSTJs can evaluate the whole relational algebra, while DSTs can evaluate the semijoin algebra which forms an important subset of the relational algebra. Furthermore, on the class of bounded degree graphs (and analogously, also for bounded degree databases), DSAs can evaluate all Boolean queries formulated in relational algebra or first-order logic with modulo counting quantifiers. In fact, on this class, we believe DSAs to be equivalent to first-order logic with modulo counting quantifiers. A direction for future research is to extend the current model with arithmetic and aggregation, as SQL queries support modest forms of counting.

Acknowledgements. We thank the anonymous referees for their helpful and inspiring comments. We also thank Jan Van den Bussche for inspiring discussions. The fourth author is supported by FWO Pegasus Marie Curie Fellowship.

References

1 F. Afrati, V. Borkar, M. Carey, N. Polyzotis, and J. Ullman. Map-reduce extensions and recursive queries. In *ICDE*, 2011.

2 F. Afrati, D. Fotakis, and J. Ullman. Enumerating subgraph instances using map-reduce. In *ICDE*, 2013.
3 F. Afrati, P. Koutris, D. Suciu, and J. Ullman. Parallel skyline queries. In *ICDT*, 2012.
4 F. Afrati, A. Dash Sarma, S. Salihoglu, and J. Ullman. Upper and lower bounds on the cost of a map-reduce computation. *PVLDB*, 6(4):277–288, 2013.
5 F. Afrati and J. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, 2010.
6 F. Afrati and J. Ullman. Transitive closure and recursive datalog implemented on clusters. In *EDBT*, 2012.
7 T. Ameloot, F. Neven, and J. Van den Bussche. Relational transducers for declarative networking. *Journal of the ACM*, 60(2):15, 2013.
8 Apache Bagel. Bagel. http://spark.apache.org/docs/0.7.3/bagel-programming-guide.html.
9 P. Beame, P. Koutris, and D. Suciu. Communication steps for parallel query processing. In *PODS*, 2013.
10 P. Beame, P. Koutris, and D. Suciu. Skew in parallel query processing. In *PODS*, 2014.
11 F. Chierichetti, R. Kumar, and A. Tomkins. Max-cover in map-reduce. In *WWW*, 2010.
12 E. Codd. A relational model of data for large shared data banks. *Communication of the ACM*, 13(6):377–387, 1970.
13 J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.
14 J. Dean and S. Ghemawat. Mapreduce: a flexible data processing tool. *Communication of the ACM*, 53(1):72–77, 2010.
15 A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a highlevel dataflow system on top of mapreduce: The pig experience. *PVLDB*, 2(2):1414–1425, 2009.
16 J. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Record*, 39(1):5–19, 2010.
17 M. Kaminski and N. Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.
18 H. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for mapreduce. In *SODA*, 2010.
19 P. Koutris and D. Suciu. Parallel evaluation of conjunctive queries. In *PODS*, 2011.
20 R. Kumar, B. Moseley, S. Vassilvitskii, and A. Vattani. Fast greedy algorithms in mapreduce and streaming. In *SPAA*, 2013.
21 S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In *SPAA*, 2011.
22 Leonid Libkin. *Elements of Finite Model Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.
23 F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Transactions on Computational Logic*, 5(3):403–435, 2004.
24 Juha Nurmonen. Counting modulo quantifiers on finite structures. *Information and Computation*, 160(1-2):62–87, 2000.
25 C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference*, 2008.
26 Apache Pig. Pig. `http://pig.apache.org/`.
27 Apache Spark. Spark. `http://spark.apache.org`.
28 Apache Spark. Spark programming guide. `http://spark.apache.org/docs/latest/programming-guide.html`.
29 S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *WWW*, 2011.
30 Y. Tao, W. Lin, and X. Xiao. Minimal mapreduce algorithms. In *SIGMOD*, 2013.

31 A. Thusoo, J. Sen Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *ICDE*, 2010.

32 L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

33 T. White. *Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale (3. ed., revised and updated)*. O'Reilly, 2012.

34 R. Xin, J. Rosen, M. Zaharia, M. Franklin, S. Shenker, and I. Stoica. Shark: Sql and rich analytics at scale. In *SIGMOD*, 2013.

35 M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

APPENDIX

A Details omitted in Section 4

A.1 Normalisation of the initial mappers

For our lower bound proofs, it will be convenient to restrict attention to *normalised* initial mappers in the following sense.

Let $\mathcal{M} = (map_1, red_1, \ldots, map_\ell, red_\ell, agg)$ be an DSA over $\mathbf{A}_k = \Sigma \times \mathbf{D}^k_\#$. Since map_1 is generic, for each edge (a,b), the multiplicity of each key-value pair in $map_1(a,b)$ is bounded by a constant. So by encoding the multiplicity of a key-value pair with symbols from Σ, and modifying the transition systems in each reducer and aggregator, we can assume that the output of the initial mapper $map_1(a,b)$ is a set, instead of a bag.

Moreover, by further extending and renaming the alphabet Σ and the states in $\mathcal{M}$, the output of the initial mapper $map_1(a,b)$ is the following *set*, for some $n \geq 1$.

$$\bigcup_{1\leq i\leq n} \left\{ \begin{array}{l} \big\langle (\alpha_i, \#^k) : (\zeta_{1,i}, \#^k) \big\rangle, \\ \big\langle (\alpha_i, \#^k) : (\eta_{1,i}, a\#^{k-1}) \big\rangle, \\ \big\langle (\alpha_i, \#^k) : (\theta_{1,i}, b\#^{k-1}) \big\rangle, \\ \big\langle (\alpha_i, \#^k) : (\lambda_{1,i}, ab\#^{k-2}) \big\rangle \end{array} \right\} \cup \bigcup_{1\leq i\leq n} \left\{ \begin{array}{l} \big\langle (\beta_i, a\#^{k-1}) : (\zeta_{2,i}, \#^k) \big\rangle, \\ \big\langle (\beta_i, a\#^{k-1}) : (\eta_{2,i}, a\#^{k-1}) \big\rangle, \\ \big\langle (\beta_i, a\#^{k-1}) : (\theta_{2,i}, b\#^{k-1}) \big\rangle, \\ \big\langle (\beta_i, a\#^{k-1}) : (\lambda_{2,i}, ab\#^{k-2}) \big\rangle \end{array} \right\}$$

$$\cup \bigcup_{1\leq i\leq n} \left\{ \begin{array}{l} \big\langle (\gamma_i, b\#^{k-1}) : (\zeta_{3,i}, \#^k) \big\rangle, \\ \big\langle (\gamma_i, b\#^{k-1}) : (\eta_{3,i}, a\#^{k-1}) \big\rangle, \\ \big\langle (\gamma_i, b\#^{k-1}) : (\theta_{3,i}, b\#^{k-1}) \big\rangle, \\ \big\langle (\gamma_i, b\#^{k-1}) : (\lambda_{3,i}, ab\#^{k-2}) \big\rangle \end{array} \right\} \cup \bigcup_{1\leq i\leq n} \left\{ \begin{array}{l} \big\langle (\delta_i, ab\#^{k-2}) : (\zeta_{4,i}, \#^k) \big\rangle, \\ \big\langle (\delta_i, ab\#^{k-2}) : (\eta_{4,i}, a\#^{k-1}) \big\rangle, \\ \big\langle (\delta_i, ab\#^{k-2}) : (\theta_{4,i}, b\#^{k-1}) \big\rangle, \\ \big\langle (\delta_i, ab\#^{k-2}) : (\lambda_{4,i}, ab\#^{k-2}) \big\rangle \end{array} \right\}$$

$$\cup \bigcup_{1\leq i\leq n} \left\{ \begin{array}{l} \big\langle (\epsilon_i, a\#^{k-1}) : (\zeta_{5,i}, \#^k) \big\rangle, \\ \big\langle (\epsilon_i, a\#^{k-1}) : (\eta_{5,i}, a\#^{k-1}) \big\rangle, \\ \big\langle (\epsilon_i, a\#^{k-1}) : (\theta_{5,i}, b\#^{k-1}) \big\rangle, \\ \big\langle (\epsilon_i, a\#^{k-1}) : (\lambda_{5,i}, ab\#^{k-2}) \big\rangle \end{array} \right\} \cup \bigcup_{1\leq i\leq n} \left\{ \begin{array}{l} \big\langle (\epsilon_i, b\#^{k-1}) : (\zeta_{6,i}, \#^k) \big\rangle, \\ \big\langle (\epsilon_i, b\#^{k-1}) : (\eta_{6,i}, a\#^{k-1}) \big\rangle, \\ \big\langle (\epsilon_i, b\#^{k-1}) : (\theta_{6,i}, b\#^{k-1}) \big\rangle, \\ \big\langle (\epsilon_i, b\#^{k-1}) : (\lambda_{6,i}, ab\#^{k-2}) \big\rangle \end{array} \right\}$$

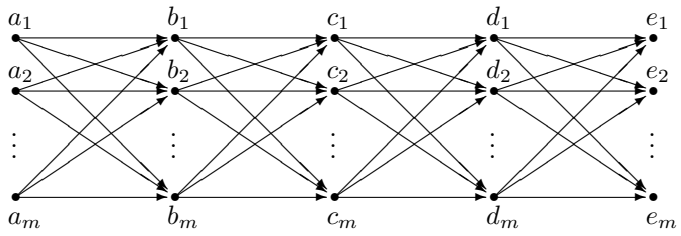
Throughout this paper, we always assume that the output of the initial mapper map_1 is always of the this form.

Note the six different types of keys, and that $(\epsilon_i, a\#^{k-1})$ and $(\epsilon_i, b\#^{k-1})$ have the same ϵ_i. Below is a brief explanation of the values for each key.

- Keys of the form $(\alpha_i, \#^k)$ will have the values that cover every edge.
- Keys of the form $(\beta_i, a\#^{k-1})$ will have the values that cover the outgoing edges of the vertex a.
- Keys of the form $(\gamma_i, a\#^{k-1})$ will have the values that cover the incoming edges of the vertex a.
- Keys of the form $(\delta_i, ab\#^{k-2})$ will have the values that cover precisely just the edge (a,b).
- Keys of the form $(\epsilon_i, a\#^{k-2})$ will have the values that cover both the incoming and outgoing edges of the vertex a.

A.2 Proof of Theorem 5

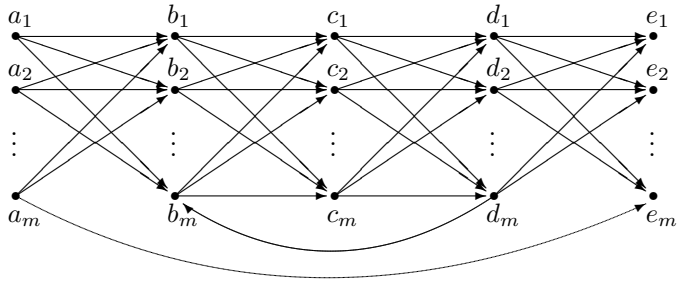
For a positive integer $m \geq 1$, we define a graph $G_m = (V_m, E_m)$ as follows.



That is, $V_m = \{a_1, \ldots, a_m, b_1, \ldots, b_m, c_1, \cdots, c_m, d_1, \ldots, d_m, e_1, \ldots, e_m\}$, and there are complete bipartite graphs with the orientation from left to right

- between the vertices $a_1, \ldots, a_m$ and $b_1, \ldots, b_m$;
- between the vertices $b_1, \ldots, b_m$ and $c_1, \ldots, c_m$;
- between the vertices $c_1, \ldots, c_m$ and $d_1, \ldots, d_m$;
- between the vertices $d_1, \ldots, d_m$ and $e_1, \ldots, e_m$.

Then, we define $\widetilde{G}_m = (V_m, \widetilde{E}_m)$ be the graph, where $\widetilde{E}_m = E_1 \setminus \{(a_m, b_m), (d_m, e_m)\} \cup \{(a_m, e_m), (d_m, b_m)\}$, as illustrated below.



Obviously for every $m \geq 1$, $G_m \notin$ TRIANGLE, while $\widetilde{G}_m \in$ TRIANGLE. The rest of this section will be devoted to the proof of Proposition 21 below, from which Theorem 5 follows immediately.

▶ **Proposition 21.** *For every DSA $\mathcal{M}$, there exists m such that $\mathcal{M}(G_m) = \mathcal{M}(\widetilde{G}_m)$.*

Let $\mathcal{M}$ be an DSA over $\mathbf{A}_k$, for some $k \geq 1$. We assume that $\mathcal{M}$ has been normalised as in Appendix A.1. Let r be the set of states and the number of registers in $\mathcal{M}$. We fix $m = r + 1$. We are going to prove that $\mathcal{M}(G_m) = \mathcal{M}(\widetilde{G}_m)$.

▶ **Claim 1.** *For each $i = 1, \ldots, \ell$, and the following holds, where $R_i = R_i(G_m)$*

- (Locality) *If there exists $t \in R_i$ that contains two vertices u and v, then $(u, v) \in E(G_m)$ and there is no other vertex in t.*
- (Vertex and edge indistinguishability) *For every $\sigma \in \Sigma$, the following equalities hold.*

(a) $\boldsymbol{\chi}_{R_i}(\sigma, a_1 \#^{k-1}) = \ldots = \boldsymbol{\chi}_{R_i}(\sigma, a_m \#^{k-1})$.
(b) $\boldsymbol{\chi}_{R_i}(\sigma, b_1 \#^{k-1}) = \ldots = \boldsymbol{\chi}_{R_i}(\sigma, b_m \#^{k-1}) = \boldsymbol{\chi}_{R_i}(\sigma, c_1 \#^{k-1}) = \ldots = \boldsymbol{\chi}_{R_i}(\sigma, c_m \#^{k-1}) = \boldsymbol{\chi}_{R_i}(\sigma, d_1 \#^{k-1}) = \ldots = \boldsymbol{\chi}_{R_i}(\sigma, d_m \#^{k-1})$.
(c) $\boldsymbol{\chi}_{R_i}(\sigma, e_1 \#^{k-1}) = \ldots = \boldsymbol{\chi}_{R_i}(\sigma, e_m \#^{k-1})$.
(d) For every edge $(u, v) \in E(G_m)$, $\boldsymbol{\chi}_{R_i}(\sigma, uv \#^{k-2})$ is the same.

Proof. The proof is by induction on i. The base case is $i = 1$. Since M is normalised, the keys in keys$(M_1(G_m))$ are as follows.

$$\begin{aligned}\text{keys}(M_1(G_m)) \quad = \quad & \{(\alpha_i, \#^k)\}_{i=1}^n \quad \cup \bigcup_{(u,v)\in E} \{(\delta_i, uv\#^{k-2})\}_{i=1}^n \\ & \cup \bigcup_{u \text{ has outgoing edge}} \{(\beta_i, u\#^{k-1})\}_{i=1}^n \quad \cup \quad \{(\epsilon_i, u\#^{k-1})\}_{i=1}^n \\ & \cup \bigcup_{u \text{ has incoming edge}} \{(\gamma_i, u\#^{k-1})\}_{i=1}^n \quad \cup \quad \{(\epsilon_i, u\#^{k-1})\}_{i=1}^n\end{aligned}$$

For the proof of locality, we are going to analyse the output $red_1(\mathsf{key}, \text{values}(\mathsf{key}, M_1(G_m)))$, for each $\mathsf{key} \in \text{keys}(M_1(G_m))$.

- For each $\mathsf{key} = (\alpha_i, \#^k)$, where $1 \le i \le n$,
$$\text{values}(\mathsf{key}, M_1(G_m)) \quad = \quad \bigcup_{(u,v)\in E} \left\{ \begin{array}{l} (\zeta_{1,i}, \#^k),\ (\eta_{1,i}, u\#^{k-1}), \\ (\theta_{1,i}, v\#^{k-1}),\ (\lambda_{1,i}, uv\#^{k-2}) \end{array} \right\}$$
Since the output of $R_1(\mathsf{key}, \text{values}(\mathsf{key}, M_1))$ has to be independent of the order of elements in values(key, M_1), and $m > r$, every output $t \in red_1(\mathsf{key}, \text{values}(\mathsf{key}, M_1))$ cannot contain any vertex.
Otherwise, suppose there is $t \in R_1(\mathsf{key}, \text{values}(\mathsf{key}, M_1))$ that contains a vertex u. Since $m > r$, we can pick a vertex v whose neighbourhood is the same as u, and not appearing in the last configuration of the transition system of red_1. That is, the vertices $a_1, \ldots, a_m$ have the same neighbourhood; and so do $b_1, \ldots, b_m$ and so do $c_1, \ldots, c_m$ and so on. Hence, if u is, say a_1, then we can pick one of the vertices $a_2, \ldots, a_m$ for v that does not appear in the last configuration of red_1. Likewise, if u is one of $b_1, \ldots, b_m, \ldots, e_1, \ldots, e_m$. Then we "swap" the occurrences of u and v. Due to the fact that red_1 is generic, now the outputs of red_1 now does not contain u, but v. This contradicts the requirement that the output of red_1 must be independent of the order of the input.
- For each $\mathsf{key} = (\beta_i, u\#^{k-1})$, where $1 \le i \le n$ and u is a vertex with outgoing edge,
$$\text{values}(\mathsf{key}, M_1(G_m)) \quad = \quad \bigcup_{(u,v)\in E} \left\{ \begin{array}{l} (\zeta_{2,i}, \#^k),\ (\eta_{2,i}, u\#^{k-1}), \\ (\theta_{2,i}, v\#^{k-1}),\ (\lambda_{2,i}, uv\#^{k-2}) \end{array} \right\}$$
In this case, u can be one of $a_1, \ldots, a_m, b_1, \ldots, b_m, c_1, \ldots, c_m, d_1, \ldots, d_m$. The outdegree of each such vertex is m.
Similar to the proof above, since the output of $red_1(\mathsf{key}, \text{values}(\mathsf{key}, M_1))$ has to be independent of the order of elements in values(key, M_1), and $m > r$, every output $t \in red_1(\mathsf{key}, \text{values}(\mathsf{key}, M_1))$ cannot contain any vertex other than u itself.
For the case of $\mathsf{key} = (\gamma_i, u\#^{k-1})$, and $\mathsf{key} = (\epsilon_i, u\#^{k-1})$, the base case can be established in a similar manner.
- For each $\mathsf{key} = (\delta_i, uv\#^{k-2})$, where $1 \le i \le n$ and (u, v) is an edge,
$$\text{values}(\mathsf{key}, M_1(G_m)) \quad = \quad \left\{ \begin{array}{l} (\zeta_{4,i}, \#^k),\ (\eta_{4,i}, v\#^{k-1}), \\ (\theta_{4,i}, u\#^{k-1}),\ (\lambda_{4,i}, vu\#^{k-2}) \end{array} \right\},$$
In this case, it is obvious that for each $t \in red_1(\mathsf{key}, \text{values}(\mathsf{key}, M_1(G_m)))$, if t contains two vertices, then these two vertices must be u and v. Since red_1 is generic, there is no other vertex in t.

This settles the locality property for the base case. The vertex and edge indistinguishability follows from the fact that

- the neighbourhoods of each of the vertices $a_1, \ldots, a_m$ are the same;
- the neighbourhoods of each of the vertices $e_1, \ldots, e_m$ are the same;
- the neighbourhoods of each of the vertices $b_1, \ldots, b_m, c_1, \ldots, c_m, d_1, \ldots, d_m$ are the same up to $\mathbf{D}$-bijection.

This settles the base case.

For the induction hypothesis, assume it holds for case i. We are going to show that it holds for case $i+1$.

Applying the induction hypothesis, every $t \in R_i(G_m)$ can contain at most two different vertices, and these two vertices are connected by an edge, and hence, (since map_{i+1} is generic) so is every value $\mathsf{val} \in \text{values}(M_{i+1}(G_m))$ and every key $\mathsf{key} \in \text{keys}(M_{i+1}(G_m))$. Therefore, there are only three types of keys in $\text{keys}(M_{i+1}(G_m))$.

- Keys that do not have contain any vertex.
  Suppose key does not contain any vertex, and let $Z = \text{values}(\mathsf{key}, M_{i+1}(G_m))$. By the induction hypothesis, and by the genericity of map_{i+1},
  - *(Locality for Z)* if there exists $\mathsf{val} \in Z$ that contains two different vertices u and v, then $(u,v) \in E(G_m)$ and there is no other vertex in val.
  - *(Vertex and edge indistinguishability for Z)* For every $\sigma \in \Sigma$, the following equalities hold.
    (a) $\boldsymbol{\chi}_Z(\sigma, a_1\#^{k-1}) = \ldots = \boldsymbol{\chi}_Z(\sigma, a_m\#^{k-1})$.
    (b) $\boldsymbol{\chi}_Z(\sigma, b_1\#^{k-1}) = \ldots = \boldsymbol{\chi}_Z(\sigma, b_m\#^{k-1}) = \boldsymbol{\chi}_Z(\sigma, c_1\#^{k-1}) = \ldots = \boldsymbol{\chi}_Z(\sigma, c_m\#^{k-1}) = \boldsymbol{\chi}_Z(\sigma, d_1\#^{k-1}) = \ldots = \boldsymbol{\chi}_Z(\sigma, d_m\#^{k-1})$.
    (c) $\boldsymbol{\chi}_Z(\sigma, e_1\#^{k-1}) = \ldots = \boldsymbol{\chi}_Z(\sigma, e_m\#^{k-1})$.
    (d) For every edge $(u,v) \in E(G_m)$, $\boldsymbol{\chi}_Z(\sigma, uv\#^{k-2})$ is the same.

  If none of the values in Z contains any vertex, then by the genericity of red_{i+1}, every output $t \in red_{i+1}(\mathsf{key}, Z)$ does not contain any vertex either.
  If there exists $\mathsf{val} \in Z$ that contains a vertex, then there are at least m values $\mathsf{val}_1, \ldots, \mathsf{val}_m$ that contain different vertices. If there exists $\mathsf{val} \in Z$ that contains an edge, then for every edge $(u,v) \in E(G_m)$, there is $\mathsf{val}' \in Z$ that contains the edge (u,v).
  Similar to the one in the base case $i = 1$, since $m > r$ and the output of $red_{i+1}(\mathsf{key}, Z)$ must be independent of the order of elements in Z, they must not contain any vertex. Hence, in this case, every output $t \in red_{i+1}(\mathsf{key}, Z)$ does not contain any vertex.
- Keys that contain a single vertex.
  Suppose key contains a vertex, say a_1, and let $Z = \text{values}(\mathsf{key}, M_{i+1}(G_m))$. Let ξ_j be the $\mathbf{D}$-bijection that maps a_1 to a_j and a_j to a_1.
  There are two further cases.

  - There is no $\mathsf{val} = (\sigma, a_1 u\#^{k-2}) \in Z$, for any vertex u.
    Then every output $t \in red_{i+1}(\mathsf{key}, Z)$ either contains the vertex a_1 itself or does not contain any vertex. So locality holds for every output $t \in red_{i+1}(\mathsf{key}, Z)$. By genericity of map_{i+1} and the induction hypothesis, $\xi_j(Z) = \text{values}(\xi_j(\mathsf{key}), M_{i+1}(G_m))$, for each $j \in [m]$. Hence, for every $t \in red_{i+1}(\mathsf{key}, Z)$,
    $$\boldsymbol{\chi}_{Y_1}(t) = \boldsymbol{\chi}_{Y_2}(\xi_2(t)) = \ldots = \boldsymbol{\chi}_{Y_m}(\xi_m(t)),$$
    where $Y_j = red_{i+1}(\xi_j(\mathsf{key}), \xi_j(M))$, and therefore, the vertex and edge indistinguishability holds for every output $t \in red_{i+1}(\mathsf{key}, Z)$.
  - There is $\mathsf{val} = (\sigma, a_1 u\#^{k-2}) \in Z$, for some vertex u.
    By the induction hypothesis, $u \in \{b_1, \ldots, b_m\}$. By the genericity of map_{i+1},
    $$1 \leq \boldsymbol{\chi}_Z(\sigma, a_1 b_1\#^{k-2}) = \boldsymbol{\chi}_Z(\sigma, a_1 b_2\#^{k-2}) = \ldots = \boldsymbol{\chi}_Z(\sigma, a_1 b_m\#^{k-2}).$$
    Since the output of $red_{i+1}(\mathsf{key}, Z)$ is independent of the order of Z and $m > r$, every output $t \in red_{i+1}(\mathsf{key}, Z)$ either contains the vertex a_1 itself or does not contain any vertex.

In a similar manner as above, the vertex and edge indistinguishability holds for every output $t \in red_{i+1}(\mathsf{key}, Z)$.

The case where key contains the other vertices $b_1, \ldots, b_m, \ldots, e_1, \ldots, e_m$ can be established in a similar manner.

- Keys that contain two different vertices.
  Let key contains two vertices u and v and $Z = \text{values}(\mathsf{key}, M_{i+1}(G_m))$ Let $s \in R_i$ be such that $\langle \mathsf{key} : \mathsf{val} \rangle \in map_{i+1}(t)$, for some val. By the genericity of map_{i+1}, u and v are inside s. By the induction hypothesis, (u, v) is an edge and s does not contain any other vertex.
  By genericity of map_{i+1}, for every $\mathsf{val} \in \text{values}(\mathsf{key}, M_{i+1}(G_m))$, if it contains two different vertices, then they must be u and v, and likewise for every output $t \in red_{i+1}(\mathsf{key}, Z)$. Therefore, locality holds for every output in $red_{i+1}(\mathsf{key}, Z)$.
  Now, let $s = (\sigma, uv\#^{k-2})$. By the induction hypothesis, $\boldsymbol{\chi}_{R_{i-1}}(\sigma, u'v'\#^{k-2}) = \boldsymbol{\chi}_{R_{i-1}}(\sigma, uv\#^{k-2})$ for every edge $(u', v') \in E(G_m)$. By the genericity of map_{i+1} and red_{i+1}, the vertex and edge indistinguishability holds for every output $t \in red_{i+1}(\mathsf{key}, Z)$.

This completes the induction step, and hence our claim. ◀

Similar claim can also be proved for $\widetilde{G}_m$ as stated below.

▶ **Claim 2.** *For each $i = 1, \ldots, \ell$, and the following holds, where $R_i = R_i(\widetilde{G}_m)$*

- (Locality) *If there exists $t \in R_i$ that contains two vertices u and v, then $(u, v) \in E(G_m)$ and there is no other vertex in t.*
- (Vertex and edge indistinguishability) *For every $\sigma \in \Sigma$, the following equalities hold.*

(a) $\boldsymbol{\chi}_{R_i}(\sigma, a_1\#^{k-1}) = \ldots = \boldsymbol{\chi}_{R_i}(\sigma, a_m\#^{k-1})$.
(b) $\boldsymbol{\chi}_{R_i}(\sigma, b_1\#^{k-1}) = \ldots = \boldsymbol{\chi}_{R_i}(\sigma, b_m\#^{k-1}) = \boldsymbol{\chi}_{R_i}(\sigma, c_1\#^{k-1}) = \ldots = \boldsymbol{\chi}_{R_i}(\sigma, c_m\#^{k-1}) = \boldsymbol{\chi}_{R_i}(\sigma, d_1\#^{k-1}) = \ldots = \boldsymbol{\chi}_{R_i}(\sigma, d_m\#^{k-1})$.
(c) $\boldsymbol{\chi}_{R_i}(\sigma, e_1\#^{k-1}) = \ldots = \boldsymbol{\chi}_{R_i}(\sigma, e_m\#^{k-1})$.
(d) For every edge $(u, v) \in E(G_m)$, $\boldsymbol{\chi}_{R_i}(\sigma, uv\#^{k-2})$ is the same.

The two claims above imply that $R_\ell(G_m)$ is precisely $R_\ell(\widetilde{G}_m)$ with the exceptions for $t \in R_\ell(G_m)$ that contains both a_m, b_m and both d_m, e_m and for $t \in R_\ell(\widetilde{G}_m)$ that contains both a_m, e_m and both d_m, b_m. That is, for each $\sigma \in \Sigma$,

$$\begin{aligned}
\boldsymbol{\chi}_{R_\ell(G_m)}(\sigma, a_m b_m) &= \boldsymbol{\chi}_{R_\ell(\widetilde{G}_m)}(\sigma, a_m e_m) \\
\boldsymbol{\chi}_{R_\ell(G_m)}(\sigma, d_m e_m) &= \boldsymbol{\chi}_{R_\ell(\widetilde{G}_m)}(\sigma, d_m c_m) \\
\boldsymbol{\chi}_{R_\ell(G_m)}(t) &= \boldsymbol{\chi}_{R_\ell(\widetilde{G}_m)}(t) \qquad \text{for all other } t
\end{aligned}$$

What is left now is to show that $agg(R_\ell(G_m)) = agg(R_\ell(\widetilde{G}_m))$. There are two cases: (i) every $t \in R_\ell(G_m)$ does not contain two different vertices, that is, every t either does not contain any vertex, or contains only one vertex; (ii) there is $t = (\sigma, uv\#^{k-2}) \in R_\ell(G_m)$ that contains two different vertices.

For case (i), by Claims 1 and 2, $R_\ell(G_m) = R_\ell(\widetilde{G}_m)$, hence $agg(R_\ell(G_m)) = agg(R_\ell(\widetilde{G}_m))$.

For case (ii), by Claim 1, (u, v) (or, (v, u)) is an edge, and that for every edge (a, b), $(\sigma, ab\#^{k-2}) \in R_\ell(G_m)$. Let T_1 be the subset of $R_\ell(G_m)$ that contains the edges $\{a_1, \ldots, a_m\} \times \{b_1, \ldots, b_m\}$ and T_2 the edges $\{d_1, \ldots, d_m\} \times \{e_1, \ldots, e_m\}$, and $T = R_\ell(G_m) - (T_1 \cup T_2)$.

Now, consider the run of agg on T_1T_2T. That is, the run of agg on the sequence where the elements of T_1 precede those of T_2, which in turn, precede those of T.

Since $m > r$ and agg is commutative, we can assume that after reading T_1, the registers of agg do not contain both a_m, b_m. Otherwise, we can rearrange the elements of T_1 such

that the registers do not contain both a_m, b_m. Similarly, we can assume that after reading T_2, the registers of *agg* do not contain both d_m, e_m.

Let T_1' be the bag obtained by replacing every b_m with e_m, and T_2' by replacing every e_m with b_m. Such $T_1'T_2'T$ is an enumeration of elements in $R_\ell(\tilde{G}_m)$. Since *agg* is deterministic, the vertices e_m and b_m are also not found in the registers of *agg* after reading T_1' and T_2', respectively. This means the last configuration in the runs of *agg* on both T_1T_2T and $T_1'T_2'T$ are the same. Therefore, $agg(R_\ell(G_m)) = agg(R_\ell(\tilde{G}_m))$. This completes our proof of Proposition 21.

A.3 Two useful lemmas on RA-reducers

In this subsection we are going to present two lemmas, which will be crucial in our establishing the hierarchy of DSAs.

Let T_1, T_2 be two finite sets of $\mathbf{A}_k = \Sigma \times \mathbf{D}_\#^k$. Recall that for a bag B, $\boldsymbol{\chi}_B(x)$ is the characteristic function of B. We say that T_1 and T_2 are equivalent, if there exists a $\mathbf{D}$-bijection ξ such that for every $t \in \mathbf{A}_k$, $\boldsymbol{\chi}_{T_1}(t) = \boldsymbol{\chi}_{T_2}(\xi(t))$.

▶ **Lemma 22.** *Let $red = (\mathcal{S}, \rho_{in}, \rho_{out})$ be an RA-reducer with r registers. Let $m \geq r+1$, and $T_1, \ldots, T_m, T$ be bags which are pairwise disjoint, and $T_1, \ldots, T_m$ are pairwise equivalent. Then the following holds.*

On input $\langle \mathsf{key} : T_1T_2\cdots T_mT \rangle$, such that key *does not contain any vertex from $T_1 \cup \cdots \cup T_m$, any output $t \in red(\mathsf{key}, T_1\cdots T_mT)$ does not contain any vertex from $T_1 \cup \cdots \cup T_m$.*

Proof. Since key does not contain any vertex from $T_1 \cup \cdots \cup T_m$, the initial configuration $\rho_{in}(\mathsf{key})$ does not either. Let $(q, \bar{u})$ be the configuration of $\mathcal{S}$ after reading $T_1 \cdots T_mT$. There are two cases.

- If $\bar{u}$ does not contain any vertex from $T_1 \cup \cdots \cup T_m$, then by genericity of ρ_{out}, every output of $\rho_{out}(q, \bar{u})$ does not either, and our lemma holds.
- Suppose $\bar{u}$ contains a vertex u from some $t \in T_i$. Because $m \geq r+1$, there is j such that $\bar{u}$ does not contain any vertex from any $t \in T_j$. Without loss of generality, assume that $i = 1$ and $j = 2$.

  Now consider the run of $\mathcal{S}$ on $T_2T_1\cdots T_mT$. Since $T_1, \ldots, T_m$ are pairwise equivalent and disjoint, and $\mathcal{S}$ is deterministic, the final configuration of $\mathcal{S}$ on $T_2T_1\cdots T_mT$ will contain a vertex from T_2, instead and does not contain any vertex from T_1. Since every output of the reducer *red* is supposed to be independent of the order of the elements presented, they cannot contain any vertex from $T_1 \cup \cdots \cup T_m$.

This completes our proof of Lemma 22. ◀

Our second lemma states that up to a certain threshold, DSA can only recognise local properties of each node, and can not distinguish different isomorphic subgraphs. To state it, we need the following notations. Let H_1 and H_2 be two disjoint and isomorphic subgraphs of G. An isomorphism ξ between H_1 and H_2 can be extended into a $\mathbf{D}_\#$-bijection where $\xi(a) = a$, for every vertex a not in H_1 and H_2. Obviously, such ξ can also be extended to $\mathbf{A}_k = \Sigma \times \mathbf{D}_\#^k$, where $\xi((\sigma, a_1\cdots a_k)) = (\sigma, \xi(a_1)\cdots\xi(a_k))$.

▶ **Lemma 23.** *Let $\mathcal{M}$ be an ℓ-round DSA, and let Q and r be its set of states and its number of registers, respectively. Let $m \geq r+1$.*

Suppose $G = (V, E)$ is an input graph in which there are m vertices $w_1, \ldots, w_m$ whose $(2^{\ell-1})$-neighbourhoods $\mathcal{N}(w_1) = (\mathcal{N}^G_{2^{\ell-1}}(w_1), w_1)$, …, $\mathcal{N}(w_m) = (\mathcal{N}^G_{2^{\ell-1}}(w_m), w_m)$ are pairwise disjoint and isomorphic. Let $\xi_{i,j}$ be the isomorphism between $\mathcal{N}(w_i)$ and $\mathcal{N}(w_j)$, where $\xi_{i,j}(w_i) = w_j$.

For every round $i \in [\ell]$, for every tuple $t \in R_i(G)$ that contains a vertex w whose (2^{i-1})-neighbourhood lies entirely inside $\mathcal{N}(w_j)$, for some $j \in [m]$, the following holds.

(1) For every vertex v in t, g-dist$^G(w,v) \leq 2^i$, and
(2) $\boldsymbol{\chi}_{R_i(G)}(t) = \boldsymbol{\chi}_{R_i(G)}(\xi_{j,j'}(t))$, for every $j' \in [m]$.

Before, we present its proof, we remark that the bound $m \geq r+1$ in Lemma 23 is the sharpest possible. For example, consider a graph which consists of r disjoint stars. Obviously the Gaifman distances between any two centres of the stars are ∞. However, it is possible for an DSA to output for every two centres, a tuple that contains them: In the first round, we can identify the centres of those stars, and mark them with a special symbol from Σ. In the second round, we send them all to the same key and the reducer will store them all in its r registers and output, for every two centres, a tuple that contains them.

Proof. (of Lemma 23) Let $\mathcal{M} = (map_1, red_1, \ldots, map_\ell, red_\ell, agg)$ be an ℓ-round DSA with r registers. Let $m \geq r+1$.

Suppose $G = (V,E)$ is the input graph in which there are m vertices $w_1, \ldots, w_m$ and their $(2^{\ell-1})$-neighbourhoods $N(w_1), \ldots, N(w_m)$ are pairwise disjoint and isomorphic. Let $\xi_{i,j}$ be the isomorphism between $\mathcal{N}(w_i)$ and $\mathcal{N}(w_j)$, where $\xi_{i,j}(w_i) = w_j$.

Let $1 \leq i \leq \ell$ and w be a vertex whose (2^i)-neighbourhood lies entirely inside $N(w_j)$, for some $1 \leq j \leq m$. Suppose $t \in R_i$ contains w. We have to prove the following.

(1) For every vertex v in t, g-dist$(w,v) \leq 2^i$,
(2) $\xi_{j,j'}(t) \in R_i$, for every $1 \leq j' \leq m$.

The proof is by induction on i. Let M_i and R_i be the output of the mapper map_i and red_i, respectively, for each $i \in [\ell]$.

The base case is $i = 1$. By the normalisation of $\mathcal{M}$,

$$\begin{aligned}\text{keys}(M_1) \;=\; & \{(\alpha_i, \#^k)\}_{i=1}^n \;\cup \bigcup_{(u,v)\in E} \{(\delta_i, uv\#^{k-2})\}_{i=1}^n \\ & \cup \bigcup_{u \text{ has outgoing edge}} \{(\beta_i, u\#^{k-1})\}_{i=1}^n \;\cup\; \{(\epsilon_i, u\#^{k-1})\}_{i=1}^n \\ & \cup \bigcup_{u \text{ has incoming edge}} \{(\gamma_i, u\#^{k-1})\}_{i=1}^n \;\cup\; \{(\epsilon_i, u\#^{k-1})\}_{i=1}^n\end{aligned}$$

To prove part (1), we are going to analyse the output $R_1(\mathsf{key}, \text{values}(\mathsf{key}, M_1))$, for all possible $\mathsf{key} \in \text{keys}(M_1)$.

- $\mathsf{key} = (\alpha_i, \#^k)$ for some $1 \leq i \leq n$.
  In this case, the set of values in values(key, M_1) is the following.
  $$\text{values}(\mathsf{key}, M_1) \;=\; \bigcup_{(u,v)\in E} \Big\{(\zeta_{1,i}, \#^k),\ (\eta_{1,i}, u\#^{k-1}),\ (\theta_{1,i}, v\#^{k-1}),\ (\lambda_{1,i}, uv\#^{k-2})\Big\}$$
  Since $N(w_1), \ldots, N(w_m)$ are pairwise disjoint and isomorphic and $m > r$, by Lemma 22, every output $t \in red_1(\mathsf{key}, \text{values}(\mathsf{key}, M_1))$ cannot contain the vertex w. Hence, in this case, the base case holds trivially.
- $\mathsf{key} = (\beta_i, u\#^{k-1})$, for some $1 \leq i \leq n$ and a vertex u.
  In this case, the set of values in values(key, M_1) is the following.
  $$\text{values}(\mathsf{key}, M_1) \;=\; \bigcup_{(u,v)\in E} \Big\{(\zeta_{2,i}, \#^k),\ (\eta_{2,i}, u\#^{k-1}),\ (\theta_{2,i}, v\#^{k-1}),\ (\lambda_{2,i}, uv\#^{k-2})\Big\}$$
  Since the output of $red_1(\mathsf{key}, \text{values}(\mathsf{key}, M_1))$ has to be generic, in this case every output $t \in red_1(\mathsf{key}, \text{values}(\mathsf{key}, M_1))$ can only contain two vertices of Gaifman distance ≤ 2. Hence, part (1) holds.

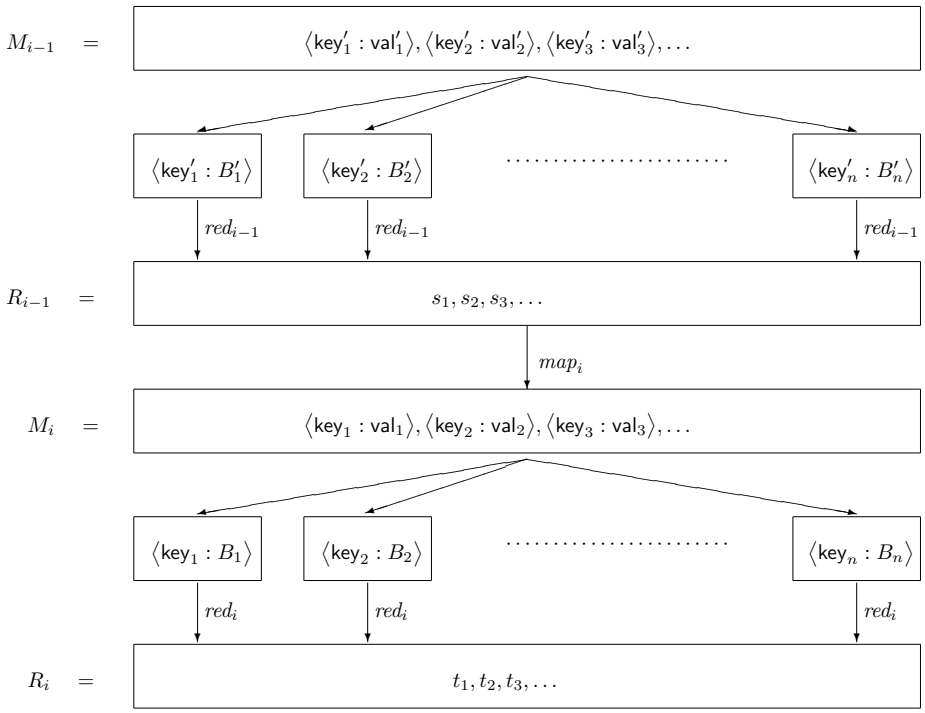


Figure 3

Part (2) also holds (trivially) because the same mapper map_1 is applied over all the edges.

- The cases of $\mathsf{key} = (\gamma_i, u\#^{k-1})$ and $\mathsf{key} = (\epsilon_i, u\#^{k-1})$ can be proved in a similar manner as the case above.
- $\mathsf{key} = (\delta_i, uv\#^{k-2})$, for some $1 \leq i \leq n$ and an edge (u, v).
  In this case, the set of values in values(key, M_1) is the following.

$$\text{values}(\mathsf{key}, M_1) \quad = \quad \Big\{(\zeta_{4,i}, \#^k),\ (\eta_{4,i}, v\#^{k-1}),\ (\theta_{4,i}, u\#^{k-1}),\ (\lambda_{4,i}, vu\#^{k-2})\Big\},$$

  That the base case holds in this case is trivial.

This establishes the lemma for the base case $i = 1$.

For the induction hypothesis, we assume that the lemma holds for $i - 1$. We are going to use the following notations.

- keys$(M_{i-1}) = \{\mathsf{key}'_1, \ldots, \mathsf{key}'_n\}$ be the keys in M_{i-1}.
- $B'_1 = \text{values}(\mathsf{key}'_1, M_{i-1})$, $B'_2 = \text{values}(\mathsf{key}'_2, M_{i-1})$ and so on.
- $R_{i-1} = \{s_1, s_2, \ldots\}$.
- keys$(M_i) = \{\mathsf{key}_1, \ldots, \mathsf{key}_n\}$ be the keys in M_i.
- $B_1 = \text{values}(\mathsf{key}_1, M_i)$, $B_2 = \text{values}(\mathsf{key}_2, M_i)$ and so on.
- $R_i = \{t_1, t_2, \ldots\}$.

See Figure 3. (For simplicity, we assume that the number of keys in M_{i-1} and M_i is the same.)

We first prove part (1). Let $t \in R_i$ contain a vertex w whose 2^i-neighbourhoods lies entirely inside some $N(w_j)$. Without loss of generality, we assume that $j = 1$. We have to prove that for every vertex x in t, the Gaifman distance g-dist$^G(w, x) \leq 2^i$.

Let key and B be such that $t \in \mathit{red}_i(\mathsf{key}, B)$. Let val_1 and val_2 be the such that

- w is contained in the pair $\langle \mathsf{key} : \mathsf{val}_1 \rangle$,
- x is contained in the pair $\langle \mathsf{key} : \mathsf{val}_2 \rangle$.

If $\mathsf{val}_1 = \mathsf{val}_2$, then we are done, because this means that there exists $s \in R_{i-1}$ that contains both x and w, which by the induction hypothesis, implies g-dist$^G(w, x) \leq 2^{i-1} < 2^i$.

So suppose that $\mathsf{val}_1 \neq \mathsf{val}_2$. Let $s_1, s_2 \in R_{i-1}$ be such that

- $\langle \mathsf{key} : \mathsf{val}_1 \rangle \in \mathit{map}_i(s_1)$ and s_1 contains w,
- $\langle \mathsf{key} : \mathsf{val}_2 \rangle \in \mathit{map}_i(s_2)$ and s_2 contains x.

Such s_1 and s_2 exists since map_i is generic, which also means that every vertex in key also appears in s_1 and s_2

There are two cases.

- key does not contain any vertex.

  Applying the induction hypothesis,

  $$\boldsymbol{\chi}_{R_{i-1}}(\xi_{1,2}(s_1)) = \ldots = \boldsymbol{\chi}_{R_{i-1}}(\xi_{1,m}(s_1)) = \boldsymbol{\chi}_{R_{i-1}}(s_1) \geq 1.$$

  If $\langle \mathsf{key} : \mathsf{val}_1 \rangle \in \mathit{map}_i(s_1)$, then since map_1 is generic, $\langle \xi_{1,j'}(\mathsf{key}) : \xi_{1,j'}(\mathsf{val}_1) \rangle \in \mathit{map}_i(\xi_{1,j'}(s_1))$. In particular,

  $$\boldsymbol{\chi}_{M_i}(\langle \xi_{1,2}(\mathsf{key}) : \xi_{1,2}(\mathsf{val}_1) \rangle) = \ldots = \boldsymbol{\chi}_{M_i}(\langle \xi_{1,m}(\mathsf{key}) : \xi_{1,m}(\mathsf{val}_1) \rangle) = \boldsymbol{\chi}_{M_i}(\langle \mathsf{key} : \mathsf{val}_1 \rangle) \geq 1.$$

  Since key does not contain any vertex, $\xi_{1,j'}(\mathsf{key}) = \mathsf{key}$, for every $j' \in [m]$. Thus,

  $$\boldsymbol{\chi}_Z(\xi_{1,2}(\mathsf{val}_1)) = \ldots = \boldsymbol{\chi}_Z(\xi_{1,m}(\mathsf{val}_1)) = \boldsymbol{\chi}_Z(\mathsf{val}_1) \geq 1,$$

  where $Z = \text{values}(\mathsf{key}, M_i)$.

  Since $N(w_1), \ldots, N(w_m)$ are pairwise disjoint and isomorphic and $m > r$, by Lemma 22, every output $\mathit{red}_i(\mathsf{key}, B)$. cannot contain the vertex w, and hence, t also must not contain w. This contradicts our assumption that t contains w.

- key contains a vertex.

  Let z be a vertex in key, which means z also appears in s_1 and s_2. By the induction hypothesis and the fact that both w and z are in s_1, we have g-dist$^G(w, z) \leq 2^{i-1}$.

  Since the 2^i-neighbourhood of w lies entirely inside $N(w_1)$, the 2^{i-1}-neighbourhood of z also lies entirely inside $N(w_1)$. Applying the induction hypothesis on s_2, the Gaifman distance g-dist$^G(z, x) \leq 2^{i-1}$ This implies g-dist$^G(w, x) \leq 2^i$.

We now prove part (2). Let $t \in R_i$ contain a vertex w whose 2^i-neighbourhoods lies entirely inside some $N(w_j)$. Without loss of generality, we assume that $j = 1$.

By the genericity of map_1, there exists $s \in R_{i-1}$ that contains w. By the induction hypothesis, for every $s \in R_{i-1}$ that contains w, the Gaifman distance from every vertex in s to w is $\leq 2^{i-1}$, and that

$$1 \leq \boldsymbol{\chi}_{R_{i-1}}(s) = \boldsymbol{\chi}_{R_{i-1}}(\xi_{1,2}(s)) = \ldots = \boldsymbol{\chi}_{R_{i-1}}(\xi_{1,m}(s))$$

Again, by genericity of map_i, for each $\langle \mathsf{key} : \mathsf{val} \rangle \in \mathit{map}_i(s)$,

$$1 \leq \boldsymbol{\chi}_{M_i}(\langle \mathsf{key} : \mathsf{val} \rangle) = \boldsymbol{\chi}_{M_i}(\xi_{1,2}(\langle \mathsf{key} : \mathsf{val} \rangle)) = \ldots = \boldsymbol{\chi}_{M_i}(\xi_{1,m}(\langle \mathsf{key} : \mathsf{val} \rangle)).$$

Let key and B be such that $t \in \mathit{red}_i(\mathsf{key}, B)$. By genericity of $\xi_{1,j'}(t) \in \mathit{red}_i(\xi_{1,j'}(\mathsf{key}), \xi_{1,j'}(B))$, for every $j' \in [m]$. In particular,

$$1 \leq \boldsymbol{\chi}_{Z_1}(t) = \boldsymbol{\chi}_{Z_2}(\xi_{1,2}(t)) = \ldots = \boldsymbol{\chi}_{Z_m}(\xi_{1,m}(t)),$$

where $Z_{j'} = red_i(\xi_{1,j'}(\mathsf{key}), \xi_{1,j'}(B))$, for every $j' \in [m]$. Summing all Z_i's for all the keys in keys(M_i),

$$1 \leq \boldsymbol{\chi}_{R_i}(t) = \boldsymbol{\chi}_{R_i}(\xi_{1,2}(t)) = \ldots = \boldsymbol{\chi}_{R_i}(\xi_{1,m}(t)).$$

This completes our proof of Lemma 23. ◀

A.4 Proof of Theorem 7

For the proof we use Nurmonen's [24] extension of the threshold equivalence version of Hanf's theorem (See [22]) for first-order logic with modulo counting quantifiers. For stating this theorem, we need the following notions of *distance*, *neighbourhoods*, and *types*.

Let $G = (V, E)$ be a directed graph. The *Gaifman distance*, denoted by g-dist$^G(u, v)$, between two vertices u and v is their distance in G when ignoring the orientation of the edges. For $r \geq 0$, the *r-ball* of a vertex v is the set $N_r^G(v)$ of vertices with Gaifman distance $\leq r$ from v. The *r-neighbourhood* of v, denoted $(\mathcal{N}_r^G(v), v)$, is the induced subgraph of G on $N_r^G(v)$, together with the designated "center" node v.

An *r-neighbourhood type* ϱ is of the form (H, u) where H is a graph and u a vertex of H such that $V(H) = N_r^H(u)$. The *degree* of $\rho = (H, u)$ is defined to be the degree of H.

We say that node v of a graph G *has r-neighbourhood type* $\varrho = (H, u)$ if there is an isomorphism from $\mathcal{N}_r^G(v)$ to H that maps v to u. For each r-neighbourhood type ϱ, we define $№_\varrho(G)$ to be the number of nodes v of G of r-neighbourhood type ϱ. For numbers $d, r, t, m \in \mathbb{N}$, the (r, t, m)-*Hanf type* of a graph G of degree $\leq d$ is a mapping τ which associates with each r-neighbourhood type ϱ of degree $\leq d$ the value $\tau(\varrho)$ defined as follows:

$(*)$ If $№_\varrho(G) < t$ then $\tau(\varrho) = №_\varrho(G)$;
otherwise $\tau(\varrho) = (t, i)$ for the number
$i \in \{0, \ldots, m-1\}$ with $№_\varrho(G) \equiv i \bmod m$.

Nurmonen's threshold equivalence version of Hanf's theorem [24] states that for every $d \geq 0$ and every sentence φ of first-order logic with modulo counting quantifiers there are numbers $r, t, m \in \mathbb{N}$ and a finite set $T_{\varphi,d}$ of (r, t, m)-Hanf types τ of degree $\leq d$, such that a graph G of degree $\leq d$ satisfies φ if, and only if, the (r, t, m)-Hanf type of G belongs to $T_{\varphi,d}$.

Proof of Theorem 7.
For given d, φ, let r, t, m and $T_{\varphi,d}$ be obtained from Nurmonen's threshold equivalence version of Hanf's theorem. Then, it suffices to construct an MRA $\mathcal{M}$ which, for an input graph G,

(1) checks if each vertex of G has degree $\leq d$, and if so,
(2) constructs, for each vertex v of G, its r-neighbourhood $(\mathcal{N}_r^G(v), v))$,
(3) computes, for each r-neighbourhood type ϱ of degree $\leq d$, the number $№_\varrho(G)$, and
(4) accepts if, and only if, the mapping τ defined as in $(*)$ belongs to $T_{\varphi,d}$.

It is straightforward to see that step (1) can be done in the 1st round of an MRA.

Concerning steps (2) and (3), note that for graphs of degree $\leq d$, the r-neighbourhood of each vertex of G contains at most d^{r+1} vertices, and there are only a fixed number f (depending on d, r) of possible r-neighbourhood types ϱ of degree $\leq d$. Thus, we can choose the alphabet Σ to contain suitable representatives of all such neighbourhood types ϱ. It is an easy (but somewhat tedious) exercise to construct an $O(\log_2 r)$-round MRA (with reducers using d^{r+1} registers), that computes a bag containing information on how often which r-neighbourhood type occurs in G.

Finally, step (4) can easily performed by an RA-aggregator which, in its state, keeps track of the number of occurrences (with exact counting up to $t-1$, and modulo m counting

for values $\geq t$) of each r-neighbourhood type, and accepts if, any only if, the according information fits to the set $T_{\varphi,d}$. ◀

A.5 Proof of part (1) of Theorem 8

We let $\Sigma = \{\mathsf{in}, \mathsf{out}, \bot\}$. We construct an ℓ-round MRA such that, on input of a graph G of degree $\leq d$, the set $R_i(G)$ consists of tuples of the form $(\bot, a, b) \in \mathbf{A}_2$, for all nodes a and b of G such that there is a walk from a to b of length 2^i.

For this, map_1 (resp., map_i for $i \geq 2$) maps each input edge (a, b) (resp., each tuple $(\bot, ab)$) to the key-value pairs $\langle \mathsf{key}_1 : \mathsf{val}_1 \rangle$ and $\langle \mathsf{key}_2 : \mathsf{val}_2 \rangle$ with $\mathsf{key}_1 = (\bot, a\#)$, $\mathsf{val}_1 = (\mathsf{in}, b\#)$ and $\mathsf{key}_2 = (\bot, b\#)$, $\mathsf{val}_2 = (\mathsf{out}, a\#)$.

For each $i \in [\ell]$, let $c_i = d^{(2^{i-1})}$. The reducer red_i for key $(\bot, v\#)$ receives as input a set of tuples $(\mathsf{out}, a_1\#), \ldots, (\mathsf{out}, a_s\#)$ and $(\mathsf{in}, b_1\#), \ldots, (\mathsf{in}, b_t\#)$ for $s, t \leq c_i$. By using $2c_i$ registers, the reducer can process its input tuples and gather a list of nodes $a_1, \ldots, a_s, b_1, \ldots, b_t$ along with information on which of the nodes corresponds to an "out" node, respectively, to an "in" node. Finally, the reducer outputs the bag of tuples of the form $(\bot, a_j b_{j'})$ for all $j \leq s$ and $j' \leq t$.

Note that the mappers and reducers are constructed in such a way that the i-th reducer outputs a tuple $(\bot, a_j b_{j'})$ iff G contains a walk of length 2^i from a_j to $b_{j'}$.

Finally, the aggregator accepts iff it receives as input at least one tuple of the form $(\bot, ab)$. In summary, we have obtained an ℓ-round DSA that accepts a graph of degree $\leq d$ iff it belongs to (2^ℓ)-WALK$_d$. It is straightforward to modify the 1st reducer such that it outputs an error message $(\mathsf{error}, \#^2)$ in case that the input graph has degree $> d$. This completes the proof of (1).

A.6 Proof of part (2) of Theorem 8

Suppose to the contrary that there is an ℓ-round DSA $\mathcal{M}$ that recognises $(2^{\ell+1})$-WALK$_2$. Let r be the number of registers in $\mathcal{M}$ and let $m = r + 1$.

Consider the graph $H = (V, E)$ in Figure 4 which consists of $2m$ components $C_1, \ldots, C_{2m}$, and $L = 2^{\ell+1}$.

Let $H' = (V, E')$ be the graph, where $E' = \Big(E \setminus \{(a_{1,L-1}, a_{1,L})\}\Big) \cup \{(b_{1,L-2}, a_{1,L})\}$. Obviously $H \in L$-WALK$_2$, but $H' \notin L$-WALK$_2$. We are going to show that $\mathcal{M}(H) = \mathcal{M}(H')$, hence, yields a contradiction that $\mathcal{G}(\mathcal{M}) = L$-WALK$_2$.

First, we observe that for every vertex v in a component, say C_1, in H, there are $2m-1$ other vertices $w_1, \ldots, w_{2m-1}$ from the components $C_2, \ldots, C_{2m}$, respectively, such that the $2^{\ell-1}$-neighbourhoods of $v, w_1, \ldots, w_m$ are disjoint and isomorphic. The same also holds for every vertex in the graph H'.

Applying Lemma 23, every $t \in R_\ell(H)$ can contains two vertices u and v, if their Gaifman distance is $\leq 2^\ell$, and likewise, for every $t \in R_\ell(H')$.

What is left is to show that $agg(R_\ell(H)) = agg(R_\ell(H'))$. We consider first $agg(R_\ell(H))$. Let $T_1, \ldots, T_{2m}$ be subsets of $R_{\ell(H)}$ that contain the vertices $a_{1,0}, \ldots, a_{m,0}, b_{1,0}, \ldots, b_{m,0}$, respectively. Similarly, let $S_1, \ldots, S_{2m}$ be subsets of $R_\ell(H)$ that contain the vertices $a_{1,L}, a_{2,L-1}, \ldots, a_{m,L-1}, b_{1,L-2}, \ldots,$ respectively.

By Lemma 23, $T_1, \ldots, T_{2m}$ are pairwise disjoint and equivalent, and so are $S_1, \ldots, S_{2m}$. Consider the run of agg on

$$T_1 \cdots T_m T_{m+1} \cdots T_{2m} S_1 \cdots S_m S_{m+1} \cdots S_{2m} T,$$

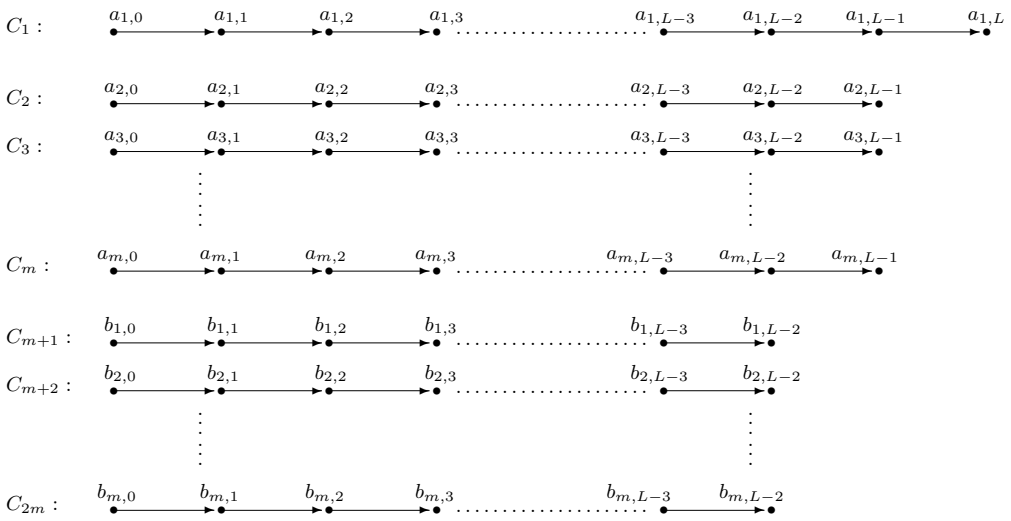


Figure 4 Illustration of the graph H in the proof of part (2) of Theorem 8

where $T = R_\ell(H) - \bigcup_{1\le i\le 2m} T_i \cup S_i$. Since $m = r+1$, and by rearranging the elements $T_1 \cdots T_{2m} S_1 \cdots S_{2m}$, we can assume that

- after reading T_m, the vertices from T_1 do not appear in the registers of *agg*;
- after reading T_{2m}, the vertices from T_{m+1} do not appear in the registers of *agg*;
- after reading S_m, the vertices from S_1 do not appear in the registers of *agg*;
- after reading S_{2m}, the vertices from S_m do not appear in the registers of *agg*.

Hence, the rest of the run of *agg* on T does not depend on the vertices from $T_1, T_{m+1}, S_1, S_{m+1}$.

Similarly, on $R_\ell(H')$, let $T'_1, \ldots, T'_{2m}$ be subsets of $R_\ell(H)$ that contain the vertices $a_{1,0}, \ldots, a_{m,0}, b_{1,0}, \ldots, b_{m,0}$, respectively; and let $S'_1, \ldots, S'_{2m}$ be subsets of $R_\ell(H)$ that contain the vertices $a_{1,N-1}, a_{2,N-1}, \ldots, a_{m,N-1}, b_{1,N-1}, b_{2,N-2} \ldots, b_{m,N-2}$, respectively, and T' be the subsets of $R_\ell(H') - \bigcup_{1\le i\le 2m} T'_i \cup S'_i$. Similar to the above, the run of *agg* on T' does not depend on $T'_1, T'_{m+1}, S'_1, S'_{m+1}$.

Since on the rest of the vertices, $R_\ell(H) = R_\ell(H')$, the outcome of *agg* on both $R_\ell(H)$ and $R_\ell(H')$ must be the same. Therefore, $agg(R_\ell(H)) = agg(R_\ell(H'))$. This completes our proof of part (2) of Theorem 8.

B Proofs omitted in Section 5

B.1 Proof of Lemma 1

We choose Σ to consist of a "dummy" symbol α and of symbols i_{in}, i_{out} for all $i \in [\ell{+}1]$, and symbols $2i$ for all $i \in [\ell]$.

We design $\mathcal{M}$ so that after each round $i \in [\ell]$, the output $R_i(G)$ contains all tuples of the form $((i{+}1)_{in}, u\#)$, where u is the endpoint of a walk of length $i{+}1$; $((i{+}1)_{out}, u\#)$, where u is the starting point of a walk of length $i{+}1$; $(2i, \#\#)$, if G contains a walk of length $2i$; and (α, uv), where (u, v) is an edge of G. The aggregator *agg* then simply checks whether its

input $R_\ell(G)$ contains at least one value of the form $(2\ell, \#\#)$. If so it outputs yes; otherwise it outputs no.

To achieve all this, the initial mapper map_1 maps every edge (u, v) of G to the key-value pairs $\langle(\alpha, u\#) : (1_{out}, \#\#)\rangle$, $\langle(\alpha, u\#) : (\alpha, uv)\rangle$, $\langle(\alpha, v\#) : (1_{in}, \#\#)\rangle$, $\langle(\alpha, v\#) : (\alpha, uv)\rangle$.

Similarly, for $i \geq 2$, the mapper map_i maps each value $(i_{in}, u\#)$ to $\langle(\alpha, u\#) : (i_{in}, \#\#)\rangle$, each value $(i_{out}, u\#)$ to $\langle(\alpha, u\#) : (i_{out}, \#\#)\rangle$, and each value (α, uv) to $\langle(\alpha, u\#) : (\alpha, uv)\rangle$ and $\langle(\alpha, v\#) : (\alpha, uv)\rangle$. (Values of the form $(2(i-1), \#\#)$ are discarded, i.e., mapped to the empty bag).

For $i \geq 1$, the reducer red_i for key $(\alpha, u\#)$ receives as input values of the form $(i_{in}, \#\#)$ if u is the endpoint of a walk of length i; $(i_{out}, \#\#)$ if u is the starting point of a walk of length i; (α, uv) if (u, v) is an edge of G, and (α, wu) if (w, u) is an edge of G. During its first pass, the reducer checks whether (a) its input contains at least one value with label i_{in}, and (b) its input contains at least one value with label i_{out}. If (a) and (b) are satisfied, in its second pass it outputs the value $(2i, \#\#)$. Furthermore, during its second pass upon reading (α, uv), it outputs the value (α, uv), and if (a) is satisfied, it also outputs the value $((i{+}1)_{in}, v\#)$. Similarly, upon reading (α, wu) it outputs the value (α, wu), and if (b) is satisfied, it also outputs the value $((i{+}1)_{out}, w\#)$.

It is straightforward to check that the resulting DST $\mathcal{M}$ recognises (2ℓ)-WALK.

B.2 Proof of Lemma 2

For positive integers $\ell, m \geq 1$, we define a graph $G_{\ell,m} = (V_{\ell,m}, E_{\ell,m})$ as illustrated in Figure 5 "plus" its transitive closure (which is not pictured in the illustration).

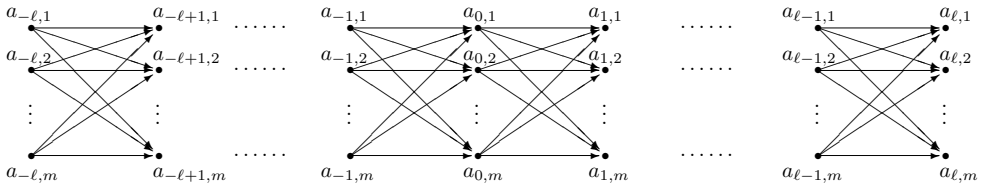


Figure 5 Illustration of the graph $G_{\ell,m}$ in the proof of Theorem 2

Formally, $G_{\ell,m} = (V_{\ell,m}, E_{\ell,m})$, where

- $V_{\ell,m} = A_{-\ell} \cup \cdots \cup A_0 \cup \cdots \cup A_\ell$, and $A_i = \{a_{i,1}, \ldots, a_{i,m}\}$ for each $-\ell \leq i \leq \ell$.
- $E_{\ell,m} = \bigcup_{-\ell \leq i < j \leq \ell} A_i \times A_j$.

Note the indexing of the sets A_i starts with $-\ell$ and ends with ℓ. For each $-\ell \leq i \leq \ell$, the outdegree of every vertex in A_i is $(\ell - i)m$, while the indegree is $(\ell + i)m$.

Obviously, for every $\ell, m \geq 1$, $G_{\ell,m} \notin (2\ell + 2)$-WALK, while $G_{\ell+1,m} \in (2\ell + 2)$-WALK. Proposition 24 below immediately implies that there is no ℓ-round DST that can recognise $(2\ell + 2)$-WALK.

▶ **Proposition 24.** *For every ℓ-round DST $\mathcal{M}$, there exists m such that $\mathcal{M}(G_{\ell,m}) = \mathcal{M}(G_{\ell+1,m})$.*

Let $\mathcal{M} = (map_1, red_1, \ldots, map_\ell, red_\ell, agg)$ be an ℓ-round DST over $\mathbf{A}_k = \Sigma \times \mathbf{D}_\#^k$, where Q and r are the set of states and the number of registers, respectively. We assume that map_1

is normalised like in the previous subsection. We pick $m = |Q|! \cdot (r+1)$. Proposition 24 can be proved via the following claim.

▶ **Claim 3.** *For each $i = 1, \ldots, \ell$, the following three properties hold, where $R_i = R_i(G_{\ell,m})$.*

- (Locality) *Every $t \in R_i$ can only contain up to two different vertices. Moreover, if $t = (\sigma, uv\#^{k-2})$, then $(u, v) \in E(G_{\ell,m})$.*
- (Vertex indistinguishability) *For every $\sigma \in \Sigma$, the following holds.*

  (1) For each $-\ell \leq j \leq -\ell + i$, for each vertex $u \in V_j$, $\boldsymbol{\chi}_{R_i}(\sigma, u\#^{k-1})$ is the same.
  (2) For each $\ell - i \leq j \leq \ell$, for each vertex $u \in V_j$, $\boldsymbol{\chi}_{R_i}(\sigma, u\#^{k-1})$ is the same.
  (3) For every vertex $u \in \bigcup_{-\ell+i+1 \leq j \leq \ell-i-1} V_j$, $\boldsymbol{\chi}_{R_i}(\sigma, u\#^{k-1})$ is the same.

  We note that on item (3), when $i = \ell$, the set $\bigcup_{-\ell+i+1 \leq j \leq \ell-i-1} V_j$ is empty.
- (Edge indistinguishability) *For every $\sigma \in \Sigma$, the following holds.*

  (1) For each $-\ell \leq j \leq -\ell+i$, for each vertex $u \in V_j$, for each edge (u, w), $\boldsymbol{\chi}_{R_i}(\sigma, uw\#^{k-2})$ is the same.
  Similarly, for each $-\ell \leq j \leq -\ell + i$, for each vertex $u \in V_j$, for each edge (w, u), $\boldsymbol{\chi}_{R_i}(\sigma, wu\#^{k-2})$ is the same.
  (2) For each $\ell - i \leq j \leq \ell$, for each vertex $u \in V_j$, for each edge (u, w), $\boldsymbol{\chi}_{R_i}(\sigma, uw\#^{k-2})$ is the same.
  Similarly, for each $\ell - i \leq j \leq \ell$, for each vertex $u \in V_j$, for each edge (w, u), $\boldsymbol{\chi}_{R_i}(\sigma, wu\#^{k-2})$ is the same.
  (3) For each vertex $u \in \bigcup_{-\ell+i+1 \leq j \leq \ell-i-1} V_j$, for each edge (u, w), $\boldsymbol{\chi}_{R_i}(\sigma, uw\#^{k-2})$ is the same.
  Similarly, for each vertex $u \in \bigcup_{-\ell+i+1 \leq j \leq \ell-i-1} V_j$, for each edge (w, u), $\boldsymbol{\chi}_{R_i}(\sigma, wu\#^{k-2})$ is the same.

Proof. In this proof, for simplicity, M_i means $M_i(G_{\ell,m})$, for each $i = 1, \ldots, \ell$. The proof is by induction on i. The base case is $i = 1$. Let $red_1 = (\mathcal{A}_1, \mathcal{T}_1, \rho_{in})$ and let $\mathsf{key} \in \text{keys}(M_1)$. By the normalisation of map_1, there are three possible cases.

- key does not contain any vertex.
  By the normalisation of map_1, if there exists $t = (\sigma, u\#^{k-1}) \in \text{values}(\mathsf{key}, M_1)$ for some vertex u, then for every vertex v, $(\sigma, v\#^{k-1}) \in \text{values}(\mathsf{key}, M_1)$. Also, if there exists $t = (\sigma, uv\#^{k-2}) \in \text{values}(\mathsf{key}, M_1)$ then (u, v) is an edge and for every edge (u', v'), $(\sigma, u'v'\#^{k-2}) \in \text{values}(\mathsf{key}, M_1)$.
  First, we show that locality holds for this case. Let T be an enumeration of values in $\mathsf{key}, M_1)$. We consider the run of $\mathcal{A}_1$ on T and suppose that q is the final state of $\mathcal{A}_1$. Now running the transducer system $\mathcal{T}_1$ on T, on reading each $\mathsf{val} \in \mathsf{key}, M_1)$, $\mathcal{T}_1$ can output only key-value pairs whose vertices are from val itself. Otherwise, it will violate the requirement that the output $red_1(\mathsf{key}, \mathsf{key}, M_1))$ is independent of the order of the input.
  By the normalisation of map_1, if t contains two different vertices, then they must be an edge. So, locality holds immediately.
- key contains only one vertex.
  Let u be the vertex in key. Every $\mathsf{val} \in \text{values}(\mathsf{key}, M_1)$ can only contain the neighbours of u. Similar to the above, suppose q is the final state of $\mathcal{A}_1$ after reading values(key, M_1). By the commutativity of red_1, on reading each $\mathsf{val} \in \text{values}(\mathsf{key}, M_1)$, the transducer system $\mathcal{T}_1$ only output t which contains vertices from val and key only. Therefore, the locality holds.

- key contains two vertices.
  Let u, v be the vertices found in key. By the genericity of map_1, (u, v) must be an edge, and the vertices in every $\mathsf{val} \in \text{values}(\mathsf{key}, M_1)$ can only be u and v.

The vertex and edge indistinguishability follows because

- the neighbourhoods of the vertices $a_{-\ell,1}, \ldots, a_{-\ell,m}$ are the same;
- the neighbourhoods of the vertices $a_{\ell,1}, \ldots, a_{\ell,m}$ are the same;
- the neighbourhoods of the vertices from $V_{\ell,m} - (V_{-\ell} \cup V_\ell)$ are the same up to $\mathbf{D}$-bijection.

Towards the induction step, we assume that the claim holds for case i. That is, the three properties: locality, vertex indistinguishability and edge indistinguishability holds for R_i.

For every $\mathsf{key} \in \text{keys}(M_{i+1})$, we define $T_{\mathsf{key}} = \{t \in R_i \mid \text{there is } \mathsf{val} \text{ such that } \langle \mathsf{key} : \mathsf{val} \rangle \in map_{i+1}(t)\}$. Since, map_{i+1} is generic, the locality, vertex and edge indistinguishability property also holds for T_{key}, for every $\mathsf{key} \in \text{keys}(M_{i+1})$. That is, the following holds.

- (Locality for T_{key}) Every $t \in T_{\mathsf{key}}$ can only contain up to two different vertices. Moreover, if $t = (\sigma, uv\#^{k-2})$, then $(u, v) \in E(G_{\ell,m})$.
- (Vertex indistinguishability for T_{key}) For every $\sigma \in \Sigma$, the following holds.

(1) For each $-\ell \leq j \leq -\ell + i$, for each vertex $u \in V_j$, $\boldsymbol{\chi}_{T_{\mathsf{key}}}(\sigma, u\#^{k-1})$ is the same.
(2) For each $\ell - i \leq j \leq \ell$, for each vertex $u \in V_j$, $\boldsymbol{\chi}_{T_{\mathsf{key}}}(\sigma, u\#^{k-1})$ is the same.
(3) For every vertex $u \in \bigcup_{-\ell+i+1 \leq j \leq \ell-i-1} V_j$, $\boldsymbol{\chi}_{T_{\mathsf{key}}}(\sigma, u\#^{k-1})$ is the same.

- (Edge indistinguishability for T_{key}) For every $\sigma \in \Sigma$, the following holds.

(1) For each $-\ell \leq j \leq -\ell+i$, for each vertex $u \in V_j$, for each edge (u, w), $\boldsymbol{\chi}_{T_{\mathsf{key}}}(\sigma, uw\#^{k-2})$ is the same.
  Similarly, for each $-\ell \leq j \leq -\ell + i$, for each vertex $u \in V_j$, for each edge (w, u), $\boldsymbol{\chi}_{T_{\mathsf{key}}}(\sigma, wu\#^{k-2})$ is the same.
(2) For each $\ell - i \leq j \leq \ell$, for each vertex $u \in V_j$, for each edge (u, w), $\boldsymbol{\chi}_{T_{\mathsf{key}}}(\sigma, uw\#^{k-2})$ is the same.
  Similarly, for each $\ell - i \leq j \leq \ell$, for each vertex $u \in V_j$, for each edge (w, u), $\boldsymbol{\chi}_{T_{\mathsf{key}}}(\sigma, wu\#^{k-2})$ is the same.
(3) For each vertex $u \in \bigcup_{-\ell+i+1 \leq j \leq \ell-i-1} V_j$, for each edge (u, w), $\boldsymbol{\chi}_{T_{\mathsf{key}}}(\sigma, uw\#^{k-2})$ is the same.
  Similarly, for each vertex $u \in \bigcup_{-\ell+i+1 \leq j \leq \ell-i-1} V_j$, for each edge (w, u), $\boldsymbol{\chi}_{T_{\mathsf{key}}}(\sigma, wu\#^{k-2})$ is the same.

The proof that the case $i + 1$ holds is very similar to the one of the base case. That is, the locality follows from the fact that the output of red_{i+1} must be independent of the order of the input and that $m \geq r + 1$. The vertex and edge indistinguishability follows from the induction hypothesis and the fact that the vertex and edge indistinguishability hold for T_{key}, for each $\mathsf{key} in \text{keys}(M_{i+1})$. This completes the induction step, and hence our proof of the claim. ◀

In a similar manner, we can prove a similar claim for $G_{\ell+1,m}$.

▶ **Claim 4.** *For each $i = 1, \ldots, \ell$, the following three properties hold, where $R_i = R_i(G_{\ell+1,m})$.*

- (Locality) *Every $t \in R_i$ can only contain up to two different vertices. Moreover, if $t = (\sigma, uv\#^{k-2})$, then $(u, v) \in E(G_{\ell,m})$.*
- (Vertex indistinguishability) *For every $\sigma \in \Sigma$, the following holds.*

(1) For each $-\ell - 1 \leq j \leq -\ell - 1 + i$, for each vertex $u \in V_j$, $\boldsymbol{\chi}_{R_i}(\sigma, u\#^{k-1})$ is the same.
(2) For each $\ell + 1 - i \leq j \leq \ell + 1$, for each vertex $u \in V_j$, $\boldsymbol{\chi}_{R_i}(\sigma, u\#^{k-1})$ is the same.

(3) *For every vertex* $u \in \bigcup_{-\ell+i \le j \le \ell-i} V_j$, $\boldsymbol{\chi}_{R_i}(\sigma, u\#^{k-1})$ *is the same.*

- (Edge indistinguishability) *For every* $\sigma \in \Sigma$, *the following holds.*

(1) *For each* $-\ell - 1 \le j \le -\ell - 1 + i$, *for each vertex* $u \in V_j$, *for each edge* (u, w), $\boldsymbol{\chi}_{R_i}(\sigma, uw\#^{k-2})$ *is the same.*
Similarly, for each $-\ell - 1 \le j \le -\ell - 1 + i$, *for each vertex* $u \in V_j$, *for each edge* (w, u), $\boldsymbol{\chi}_{R_i}(\sigma, wu\#^{k-2})$ *is the same.*

(2) *For each* $\ell - i + 1 \le j \le \ell + 1$, *for each vertex* $u \in V_j$, *for each edge* (u, w), $\boldsymbol{\chi}_{R_i}(\sigma, uw\#^{k-2})$ *is the same.*
Similarly, for each $\ell + 1 - i \le j \le \ell + 1$, *for each vertex* $u \in V_j$, *for each edge* (w, u), $\boldsymbol{\chi}_{R_i}(\sigma, wu\#^{k-2})$ *is the same.*

(3) *For each vertex* $u \in \bigcup_{-\ell+i \le j \le \ell-i} V_j$, *for each edge* (u, w), $\boldsymbol{\chi}_{R_i}(\sigma, uw\#^{k-2})$ *is the same.*
Similarly, for each vertex $u \in \bigcup_{-\ell+i \le j \le \ell-i} V_j$, *for each edge* (w, u), $\boldsymbol{\chi}_{R_i}(\sigma, wu\#^{k-2})$ *is the same.*

What is left to prove is that $agg(R_\ell(G_{\ell,m})) = agg(R_\ell(G_{\ell+1,m}))$. We first make the following observation.

Let $\xi : G_{\ell,m} \to G_{\ell+1,m}$ be a graph homomorphism, where

- ξ maps $A_0(G_{\ell,m})$ to $A_0(G_{\ell,m})$;
- ξ maps $A_i(G_{\ell,m})$ to $A_{i+1}(G_{\ell+1,m})$, for each $i = 2, \ldots, \ell$;
- ξ maps $A_i(G_{\ell,m})$ to $A_{i-1}(G_{\ell+1,m})$, for each $i = -2, \ldots, -\ell$.

The following claim shows that $R_\ell(G_{\ell,m})$ and $R_\ell(G_{\ell+1,m})$ are essentially the "same."

▶ **Claim 5.** *For each* $i = 1, \ldots, \ell$, *for every* $t \in \mathbf{A}_k$, *either*

- $\boldsymbol{\chi}_{R_i(G_{\ell+1,m})}(\xi(t)) = \boldsymbol{\chi}_{R_i(G_{\ell,m})}(t)$
- $\boldsymbol{\chi}_{R_i(G_{\ell,m})}(t) \ge m$ *and* $\boldsymbol{\chi}_{R_i(G_{\ell+1,m})}(\xi(t)) - \boldsymbol{\chi}_{R_i(G_{\ell,m})}(t)$ *is a multiple of* m.

Proof. The proof follows from the fact that each mapper and reducer are generic; and Claims 3 and 4 and the fact that the number of edges in $|E(G_{\ell+1,m})| - |E(G_{\ell,m})|$ is a multiple of m. The claim then can be proved by a straightforward induction on i. ◀

Now we are going to show that $agg(R_\ell(G_{\ell,m})) = agg(R_\ell(G_{\ell+1,m}))$. Let T_0, T_1, T_2 be subsets of $R_\ell(G_{\ell,m})$ of values that contain no vertex, one vertex and two vertices, respectively; and T_0', T_1', T_2' be subsets of $R_\ell(G_{\ell+1,m})$ of values that contain no vertex, one vertex and two vertices, respectively.

By Claim 5, for every $t \in T_0'$, $\boldsymbol{\chi}_{T_0'}(t) = \boldsymbol{\chi}_{T_0}(t)$, or $\boldsymbol{\chi}_{T_0'}(t) - \boldsymbol{\chi}_{T_0}(t)$ is a multiple of m and $\boldsymbol{\chi}_{T_0}(t) \ge m$. Hence, since agg is deterministic, and m is a factor of $|Q|!$, it will end in the same state after reading T_0 and T_0'. Next, we are going to show that

- after reading T_0T_1 and $T_0'T_1'$, the aggregator agg has the same configuration;
- after reading $T_0T_1T_2$ and $T_0'T_1'T_2'$, the aggregator agg has the same configuration.

Note that $|V(G_{\ell+1}, m)| = |V(G_{\ell,m})| + 2m$, hence, $|T_1'| = |T_1| + K \times 2m$, for some constant $1 \le K \le |\Sigma|$. Since $|T_1| > |Q|$, agg will enter into a cycle (of length $\le |Q|$). Now, agg is deterministic and m is a factor of $|Q|!$. So agg will also exit T_1' in the same state. That the content of the registers are also the same can be obtained by rearranging the input. Hence, after reading T_0T_1 and $T_0'T_1'$, the aggregator agg has the same configuration.

In a similar manner, we can show that after reading $T_0T_1T_2$ and $T_0'T_1'T_2'$. This completes our proof that $agg(R_\ell(G_{\ell,m})) = agg(R_\ell(G_{\ell+1,m}))$, and therefore, Proposition 24.

B.3 Proof of Theorem 13

The structure of the proof is almost the same as in the proof of Theorem 5. However, we need a slightly different class of graphs.

For a positive integer $m \geq 1$, we define a graph $H_m = (V_m, E_m)$ as illustrated in Figure 6.

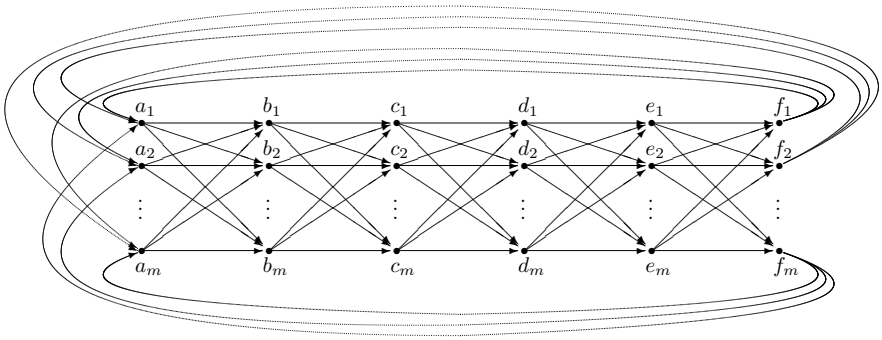


Figure 6 Illustration of the graph H_m in the proof of Theorem 13

That is, $V_m = \{a_1, \ldots, a_m, b_1, \ldots, b_m, c_1, \cdots, c_m, d_1, \ldots, d_m, e_1, \ldots, e_m, f_1, \ldots, f_m\}$, and there are complete bipartite graph with the orientation from left to right

- between the vertices $a_1, \ldots, a_m$ and $b_1, \ldots, b_m$;
- between the vertices $b_1, \ldots, b_m$ and $c_1, \ldots, c_m$;
- between the vertices $c_1, \ldots, c_m$ and $d_1, \ldots, d_m$;
- between the vertices $d_1, \ldots, d_m$ and $e_1, \ldots, e_m$;
- between the vertices $e_1, \ldots, e_m$ and $f_1, \ldots, f_m$;
- between the vertices $f_1, \ldots, f_m$ and $a_1, \ldots, a_m$.

The difference between G_m and H_m is the "going back" edges from $f_1, \ldots, f_m$ to $a_1, \ldots, a_m$.

Then, we define $\widetilde{H}_m = (V_m, \widetilde{E}_m)$ be the graph, where $\widetilde{E}_m = E_1 \setminus \{(b_m, c_m), (e_m, f_m)\} \cup \{(b_m, f_m), (e_m, c_m)\}$, as illustrated in Figure 7.

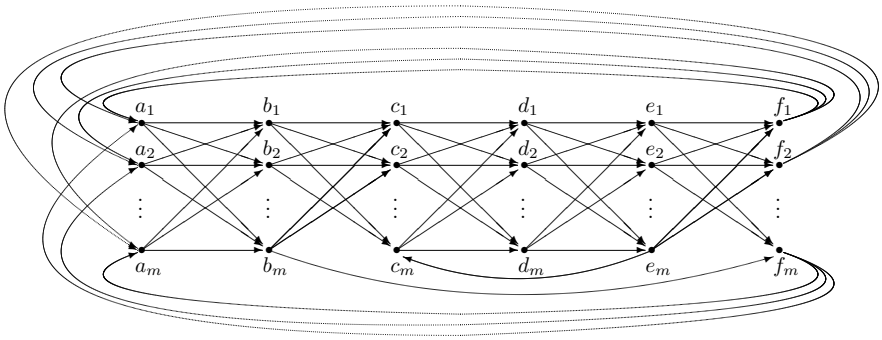


Figure 7 Illustration of the graph $\widetilde{H}_m$ in the proof of Theorem 13

Obviously for every $m \geq 1$, $H_m \notin$ TRIANGLE, while $\widetilde{H}_m \in$ TRIANGLE.

▶ **Proposition 25.** *For every DST $\mathcal{M}$, there exists m such that $\mathcal{M}(H_m) = \mathcal{M}(\widetilde{H}_m)$.*

The proof is very similar to the proof of Proposition 24. It is based on the fact that for every vertex u, its neighbourhoods in both H_m and $\tilde{H}_m$ are the same. This, with the fact that $m \geq r+1$, by just looking at the u and its neighbourhood, the DST $\mathcal{M}$ cannot differentiate whether u is a vertex in H_m or $\tilde{H}_m$.

The formal proof is via the following two claims. Their proofs are rather similar to the ones in Appendix B.2.

▶ **Claim 6.** *For each $i = 1, \ldots, \ell$, the following holds, where $R_i = R_i(H_m)$.*

- (Locality) *If there exist $t \in R_i(H_m)$ that contains both u and v, then $(u, v) \in E(H_m)$ and there is no other vertex in t.*
- (Vertex indistinguishability) *For each $\sigma \in \Sigma$, for each vertex $u \in V(H_m)$, $\boldsymbol{\chi}_{R_i}(\sigma, u\#^{k-1})$ is the same.*
- (Edge indistinguishability) *For each $\sigma \in \Sigma$, for each edge $(u, v) \in E(H_m)$, $\boldsymbol{\chi}_{R_i}(\sigma, uv\#^{k-1})$ is the same.*

Proof. The proof is by induction on i. The base case is $i = 1$. As before, we assume that the initial mapper map_1 is normalised, and hence, there are three possible types of keys.

- key does not contain any vertex.
  Like in the proof of Claim 3 in Appendix B.2, the locality holds.
- key contains only one vertex.
  Let u be the vertex in key. Every val $\in$ values(key, M_1) can only contain the neighbours of u. By the commutativity of red_1 and $m \geq r+1$, every output $t \in R_1(\mathsf{key}, \text{values}(\mathsf{key}, M_1))$ must be an edge adjacent to u.
- key contains two vertices.
  Let u, v be the vertices found in key. By the genericity of map_1, (u, v) must be an edge, and the vertices in every val $\in$ values(key, M_1) can only be u and v.

The vertex and edge indistinguishability hold trivially here, because the neighbourhood of every two vertices u and v are isomorphic.

The induction step can be proved in a similar manner as in Appendix B.2. ◀

In a similar manner, we can prove the following claim.

▶ **Claim 7.** *For each $i = 1, \ldots, \ell$, the following holds, where $R_i = R_i(\widetilde{H}_m)$.*

- (Locality) *If there exist $t \in R_i(H_m)$ that contains both u and v, then $(u, v) \in E(H_m)$ and there is no other vertex in t.*
- (Vertex indistinguishability) *For each $\sigma \in \Sigma$, for each vertex $u \in V(H_m)$, $\boldsymbol{\chi}_{R_i}(\sigma, u\#^{k-1})$ is the same.*
- (Edge indistinguishability) *For each $\sigma \in \Sigma$, for each edge $(u, v) \in E(H_m)$, $\boldsymbol{\chi}_{R_i}(\sigma, uv\#^{k-1})$ is the same.*

The two claims above imply that $R_\ell(G_m)$ is precisely $R_\ell(\widetilde{G}_m)$ with the following exception. For each $\sigma \in \Sigma$,

$$\begin{aligned}
\boldsymbol{\chi}_{R_\ell(H_m)}(\sigma, b_m c_m) &= \boldsymbol{\chi}_{R_\ell(\widetilde{H}_m)}(\sigma, b_m f_m) \\
\boldsymbol{\chi}_{R_\ell(H_m)}(\sigma, e_m f_m) &= \boldsymbol{\chi}_{R_\ell(\widetilde{H}_m)}(\sigma, e_m c_m) \\
\boldsymbol{\chi}_{R_\ell(H_m)}(t) &= \boldsymbol{\chi}_{R_\ell(\widetilde{H}_m)}(t) \qquad \text{for all other } t
\end{aligned}$$

That $agg(R_\ell(G_m)) = agg(R_\ell(\widetilde{G}_m))$ can be showed in a similar manner as in Appendix A.2.

C Details omitted in Section 6

C.1 Proof of Lemma 16

The DSTJ $\mathcal{M} = (map_1, red_1, map_2, red_2, agg)$ that accepts TRIANGLE is defined as follows.

The first round. The first mapper map_1 maps each edge (u, v) to the set

$$\Big\{\big\langle(\alpha, u\#) : (\alpha_{to}, v\#)\big\rangle,\ \big\langle(\alpha, v\#) : (\alpha_{from}, u\#)\big\rangle,\ \big\langle(\alpha, uv) : (\alpha, uv)\big\rangle\Big\}$$

Intuitively, if a value $(\alpha_{to}, v\#)$ is associated with a key $(\alpha, u\#)$, it means there is an edge from u to v. Similarly, if a value $(\alpha_{from}, v\#)$ is associated with a key $(\alpha, u\#)$, it means there is an edge from v to u. It outputs the pair $\big\langle(\alpha, uv) : (\alpha, uv)\big\rangle$ in order to pass the edges to the next round.

The first reducer red_1 does the following. On each key $(\alpha, u\#)$, it uses a joiner to perform the Cartesian product between the vertices under the label α_{from} and the vertices under the label α_{to}, and label the output with β. Formally, one key $(\alpha, u\#)$, it outputs the following.

$$\{\!\{(\beta, uvw) \mid (\alpha_{from}, v\#) \text{ and } (\alpha_{to}, w\#) \text{ are associated with the key } (\alpha, u\#)\}\!\}.$$

On key (α, uv), it outputs (α, uv).

The second round. The second mapper map_2 sends (α, uv) to $\big\langle(\alpha, uv) : (\alpha, uv)\big\rangle$ and (β, uvw) to $\big\langle(\alpha, wv) : (\beta, vw)\big\rangle$. So, for each edge (u, v), there is a key $(\alpha, uv) \in M_2$, which contains the value (α, uv). Moreover, it also contains (β, vu), if there is path from v to u of length 2.

The second reducer does the following. On each key (α, uv), if red_2 sees the values (α, uv) and (β, vu), it outputs $(\mathsf{yes}, \#\#)$. Otherwise, it outputs nothing.

The aggregator. The aggregator accepts R_2, if it contains $(\mathsf{yes}, \#\#)$. It is straightforward that $\mathcal{G}(\mathcal{M}) = \text{TRIANGLE}$.

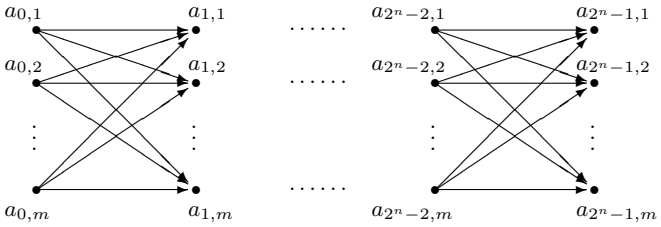
C.2 Proof of part (1) of Lemma 18

Let $\ell \geq 1$. Let G be the input graph. Like in the first round of the DSTJ in Appendix C.1, using joiner, on each round $i \in [\ell - 1]$, we can collect all pairs (u, v), where there is a walk of length 2^i from u to v.

The last round works like in the second round of the DSTJ for TRIANGLE in Appendix C.1. For each pair (u, v) in which there is walk of length 2^i, it checks whether there is a walk from v to u. If so, it outputs $(\mathsf{yes}, \#^k)$. The aggregator accepts G, it sees at least one $(\mathsf{yes}, \#^k)$.

C.3 Proof of part (2) of Lemma 18

For positive integers $n, m \geq 1$, we define a graph $G_{n,m} = (V_{n,m}, E_{n,m})$ as the following graph "plus" the edges (which are not shown in the picture) from the vertices $a_{2^n-1,1}, \ldots, a_{2^n-1,m}$ to $a_{0,1}, \ldots, a_{0,m}$.



Formally, $G_{n,m} = (V_{n,m}, E_{n,m})$, where

- $V_{n,m} = A_0 \cup \cdots \cup A_{2^n-1}$, and $A_i = \{a_{i,1}, \ldots, a_{i,m}\}$ for each $0 \leq i \leq 2^n - 1$.
- $E_{n,m} = (A_{2^n-1} \times A_0) \cup \bigcup_{0 \leq i \leq 2^n-1} A_i \times A_{i+1}$.

Obviously, $G_{\ell+2,m} \notin 2^{\ell+1}$-CYCLE, while $G_{\ell+1,m} \in 2^{\ell+1}$-CYCLE. Proposition 26 below immediately implies that there is no ℓ-round DSTJ that can recognise $2^{\ell+1}$-CYCLE.

▶ **Proposition 26.** *For every ℓ-round DSTJ $\mathcal{M}$, there exists m such that $\mathcal{M}(G_{\ell+2,m}) = \mathcal{M}(G_{\ell+1,m})$.*

We devote the rest of this subsection to our proof of Proposition 26. We fix an ℓ-round DSTJ $\mathcal{M}$ over $\mathbf{A}_k$, for some $k \geq 1$. Let r be the number of registers in $\mathcal{M}$. In the following we let $m = r + 1$. We assume that $\mathcal{M}$ has been normalised as in Appendix A.1. Moreover, as shown in the proof of Proposition 24, we can assume that on "constant" keys, i.e. keys that do not contain any vertex, the reducers output nothing.

Let $t = (\gamma, a_1 \cdots a_k) \in \mathbf{A}_k$ and let u be a vertex that appears in $a_1 \cdots a_r$. For a vertex v, we denote by $t[u \to v]$ the tuple $(\gamma, b_1 \cdots b_k)$, where $b_1 \cdots b_k$ is $a_1 \cdots a_k$ but substituting u with v.

▶ **Claim 8.** *For every $n \geq \ell + 1$, for each $i = 1, \ldots, \ell$, the following holds.*

(P1) For every $t \in R_i(G_{n,m})$, if t contains two vertices u and v, then the Gaifman distance $g\text{-}dist_{G_{n,m}}(u, v) \leq 2^i$.

(P2) For every $t \in R_i(G_{n,m})$, if t contains a vertex $a_{i,j} \in V_{n,m}$, then $t[a_{i,j} \to a_{i,j'}] \in R_i(G_{n,m})$, for every $j' = 1, \ldots, m$.

Proof. The proof is by induction on i. The basis is $i = 1$. Let $t \in R_1(G_{n,m})$. Thus, there exists key such that t is output by red_1 on the values in M_1 associated with key.

There are two possibilities for key.

- key $= (\gamma, u\#^{k-1})$ for some vertex u.

  Due to genericity of map_1, the tuples in values(key, $M_1(G_{\ell,m})$) contain only vertices adjacent to u. By the genercity of red_1, t contains only vertices that are in the 1-neighbourhood of u. Thus, if t contains two vertices, their Gaifman distance is at most two. Therefore, (P1) holds.

  That (P2) holds is as follows. Let $e \in E_{n,m}$ be an edge that contains $a_{i,j}$. For every vertex $a_{i,j'}$, we have

$$e[a_{i,j} \to a_{i,j'}] \in E_{n,m}.$$

So, if $\mathsf{val}_1, \ldots, \mathsf{val}_s$ are the values associated with key in $M_1(G_n, m)$, by the genericity of map_1, so are

$$\langle \mathsf{key}[a_{i,j} \to a_{i,j'}] : \mathsf{val}_1[a_{i,j} \to a_{i,j'}] \rangle, \ldots, \langle \mathsf{key}[a_{i,j} \to a_{i,j'}] : \mathsf{val}_s[a_{i,j} \to a_{i,j'}] \rangle \in M_1(G_{n,m}).$$

By genericity of red_1, we have $t[a_{i,j} \to a_{i,j'}] \in R_1(G_{n,m})$, and thus, (P2) holds.

- $\mathsf{key} = (\gamma, uv\#^{k-1})$.
  Due to genericity of map_1, (u, v) must be an edge, and the tuples in values($\mathsf{key}, M_1(G_{\ell,m})$) can contain only u, v. Thus, t can contain only u, v, and (P1) holds trivially. That (P2) holds is similar reasoning as above.

For the induction step, we assume that (P1) and (P2) hold for $i-1$. We are going to show that they hold for i. Let $t \in R_i(G_{n,m})$. Thus, there exists key such that t is output by red_i on the values in M_i associated with key.

Let u be a vertex contained in key. By the induction hypothesis that (P1) holds for $R_{i-1}(G_{n,m})$, and by genericity of map_i, every vertex contained in a value associated with key has Gaifman distance to u at most 2^{i-1}. This means that every vertex contained in t has Gaifman distance at most 2^{i-1} to u, thus, implying that the Gaifman distance between every two vertices in t is at most 2^i. This proves (P1).

To prove (P2), suppose t contains a vertex $a_{i,j}$. Let $\mathsf{val}_1, \ldots, \mathsf{val}_s$ be the values associated with key in $M_i(G_{n,m})$ and let $t_1, \ldots, t_s \in R_{i-1}(G_{n,m})$ where $\langle \mathsf{key} : \mathsf{val}_j \rangle \in map_i(t_j)$. By the induction hypothesis, $t_l[a_{i,j} \to a_{i,j'}] \in R_{i-1}(G_{n,m})$.

By the genericity of map_i,

$$\langle \mathsf{key}[a_{i,j} \to a_{i,j'}] : \mathsf{val}_1[a_{i,j} \to a_{i,j'}] \rangle, \ldots, \langle \mathsf{key}[a_{i,j} \to a_{i,j'}] : \mathsf{val}_s[a_{i,j} \to a_{i,j'}] \rangle \in M_i(G_{n,m}).$$

By genericity of red_i, we have $t[a_{i,j} \to a_{i,j'} \in R_i(G_{n,m})$. ◀

Let $n \geq \ell + 1$ and $0 \leq h \leq 2^n$. Let ξ be an automorphism on $G_{n,m}$, where $\xi(a_{i,j}) = a_{i+h \mod 2^n, j}$. Obviously ξ can be extended to $\mathbf{A}_k$, where $\xi(\gamma, a_1 \cdots a_k) = (\gamma, \xi(a_1) \cdots \xi(a_k))$ and $\xi(\#) = \#$.

Next we observe the following claim.

▶ **Claim 9.** *Then, for every $i = 1, \ldots, \ell$, for every $t \in R_i(G_{n,m})$, we also have $\xi(t) \in R_i(G_{n,m})$.*

Proof. The proof is by straightforward induction on i and follows immediately from the genericity of the mappers and reducers in $\mathcal{M}$, thus, omitted. ◀

Now we are going to use Claim 8 for $n = \ell + 1$ and $n = \ell + 2$ to show that $\mathcal{M}(G_{\ell+1,m}) = \mathcal{M}(G_{\ell+2,m})$. For ease of presentation, we assume that the vertices in $G_{\ell+1,m}$ are denoted by $a_{i,j}$'s, while those in $G_{\ell+2,m}$ by $b_{i,j}$'s.

We let ρ to be the graph homomorphism from $G_{\ell+2,m}$ to $G_{\ell+1,m}$, where

$$\rho(b_{i,j}) \quad = \quad \begin{cases} a_{i,j} & \text{if } 0 \leq i \leq 2^{\ell+1} - 1 \\ a_{i-2^{\ell+1},j} & \text{if } 2^{\ell+1} \leq i \leq 2^{\ell+2} - 1 \end{cases}$$

Abusing the notation, we use ρ to denote its extension to $\mathbf{A}_k$.

We are going to show that $agg(R_\ell(G_{\ell+1,m})) = agg(R_\ell(G_{\ell+2,m}))$. Observe also that Claim 9 implies for every $t \in R_\ell(G_{\ell+2,m})$, $\rho(t) \in R_\ell(G_{\ell+2,m})$. Now, since $m > r$ and agg is commutative, by similar reasoning as in Appendix B.2, $agg(R_\ell(G_{\ell+1,m})) = agg(R_\ell(G_{\ell+2,m}))$.

D Details omitted in Section 7

D.1 Semijoin algebra and relational algebra

Relational algebra (RA) expressions are defined inductively as follows:

- (Atomic expression) Every relation symbol R of arity n (for $n \geq 0$) is an RA-expression of arity n.
- (Union, intersection and difference) If e_1 and e_2 are RA-expressions of arity n (for $n \geq 0$), then so are $(e_1 \cup e_2)$, $(e_1 \cap e_2)$, and $(e_1 - e_2)$.
- (Selection) If e is an RA-expression of arity n and $i, j \in \{1, \ldots, n\}$, then $\sigma_{i=j}(e)$ is an RA-expression of arity n.
- (Projection) If e is an RA-expression of arity n, $m \geq 0$ and $i_1, \ldots, i_m \in \{1, \ldots, n\}$, then $\pi_{i_1,\ldots,i_m}(e)$ is an SJ-expression of arity m.
- (Semijoin) If e_1 and e_2 are RA-expressions of arity n_1 and n_2, respectively, and θ is of the form $\bigwedge_{s=1}^{t}(i_s{=}j_s)$ for $t \geq 1$ and $i_1, \ldots, i_t \in [n_1]$ and $j_1, \ldots, j_t \in [n_2]$, then $(e_1 \ltimes_\theta e_2)$ is an RA-expression of arity n_1.
- (Join) If e_1 and e_2 are RA-expressions of arity n_1 and n_2, respectively, and θ is of the form $\bigwedge_{s=1}^{t}(i_s{=}j_s)$ for $t \geq 1$ and $i_1, \ldots, i_t \in [n_1]$ and $j_1, \ldots, j_t \in [n_2]$, then $(e_1 \bowtie_\theta e_2)$ is an RA-expression of arity $n_1 + n_2 - t$.

Semijoin algebra expressions are relational algebra expressions in which no join operation occurs. The semantics of relational algebra expressions is as defined follows.

On a relational database DB, an relational algebra expression e defines a set of tuples $e(\mathsf{DB})$ as follows:

- If $e = R$ then $e(\mathsf{DB}) = \{\bar{a} : R(\bar{a}) \in \mathsf{DB}\}$.
- If $e = (e_1 * e_2)$ for $* \in \{\cup, -, \cap\}$ then $e(\mathsf{DB}) = e_1(\mathsf{DB}) * e_2(\mathsf{DB})$.
- If $e = \sigma_{i=j}(e')$ then $e(\mathsf{DB}) = \{\bar{a} \in e'(\mathsf{DB}) : a_i = a_j\}$.
- If $e = \pi_{i_1,\ldots,i_m}(e')$ then $e(\mathsf{DB}) = \{(a_{i_1}, \ldots, a_{i_m}) : \bar{a} \in e'(\mathsf{DB})\}$.
- If $e = (e_1 \ltimes_\theta e_2)$ then $e(\mathsf{DB}) = \{\bar{a} \in e_1(\mathsf{DB}) : \exists\, \bar{b} \in e_2(\mathsf{DB})$ with $\theta(\bar{a}, \bar{b})$ holds$\}$, where for $\theta = \bigwedge_{s=1}^{t}(i_s{=}j_s)$ the statement $\theta(\bar{a}, \bar{b})$ is satisfied iff $a_{i_s}{=}b_{j_s}$ is true for all $s \in [t]$.
- If $e = (e_1 \bowtie_\theta e_2)$ then $e(\mathsf{DB}) = \{(\bar{a}, \pi_I(\bar{b})) : \bar{a} \in e_1(\mathsf{DB}) \wedge \bar{b} \in e_2(\mathsf{DB})$ with $\theta(\bar{a}, \bar{b})$ holds$\}$, where $\pi_I(\bar{b})$ denote the projection of $\bar{b}$ to the indices in I and $I = \{1, \ldots, n_2\} - \{j \mid \exists\, i$ s.t. $(i, j) \in \theta\}$ and the statement $\theta(\bar{a}, \bar{b})$ is as above.

D.2 Proof of Part (1) of Theorem 20

We are going to show that on bounded degree databases, each RA operation can be simulated by one round DSA $\mathcal{M} = (map, red)$, in which the tuples output by the reducer has the same label T. Its generalisation for arbitrary RA-expression can be established via straightforward induction. Note that the bounded degree is only needed for the semijoin and join operations.

Union: $R \cup S$**.** The mapper works as follows.

$$map(t) \quad = \quad \begin{cases} \langle T(\bar{a}) : R(\bar{a}) \rangle & \text{if } t \text{ is } R(\bar{a}) \\ \langle T(\bar{a}) : S(\bar{a}) \rangle & \text{if } t \text{ is } S(\bar{a}) \\ \emptyset & \text{otherwise} \end{cases}$$

The reducer *red* works as follows. On key t, it outputs t itself.

Intersection: $R \cup S$**.** The mapper works like in the case $R \cup S$. The reducer *red* works as follows. On key t, it checks whether there are two tuples, one with label R and another with label S. If so, it outputs t itself. Otherwise, it outputs nothing.

Difference: $R - S$**.** The mapper works like in the case $R \cup S$. The reducer *red* works as follows. On key t, it checks whether there is a tuple with label R and there is no tuple with label S. If so, it outputs t itself. Otherwise, it outputs nothing.

Selection: $\sigma_{i=j}(R)$**.** The mapper works as follows.

$$map(t) \quad = \quad \begin{cases} \langle T(\bar{a}) : T(\bar{a}) \rangle & \text{if } t \text{ is } R(\bar{a}) \text{ and } a_i = a_j \\ \emptyset & \text{otherwise} \end{cases}$$

The reducer *red* works as follows. On key t, it outputs t itself.

Projection: $\pi_{i_1,\ldots,i_m}(R)$**.** The mapper works as follows.

$$map(t) \quad = \quad \begin{cases} \langle T(a_{i_1}, \ldots, a_{i_m}) : T(a_{i_1}, \ldots, a_{i_m}) \rangle & \text{if } t \text{ is } R(\bar{a}) \\ \emptyset & \text{otherwise} \end{cases}$$

The reducer *red* works as follows. On key t, it outputs t itself.

Semijoin: $R \ltimes_\theta S$**.** Let I and J be the projection of θ to its first and second coordinates. The mapper works as follows.

$$map(t) \quad = \quad \begin{cases} \langle T(\pi_I(\bar{a})) : R(\bar{a})) \rangle & \text{if } t \text{ is } R(\bar{a}) \\ \langle T(\pi_J(\bar{a})) : S(\bar{a})) \rangle & \text{if } t \text{ is } S(\bar{a}) \\ \emptyset & \text{otherwise} \end{cases}$$

The reducer *red* works as follows. On key t, it passes through its input, while remembering all the R-facts in its registers. Since the input dataabse DB is of bounded degree, say $\leq d$, the number R-facts associated with one particular key is also bounded by d. So we can choose the number of registers in *red* to be kd, where k is the arity of R, to accommodate all the R-facts and S-facts. If there is at least an S-fact among its input, it outputs all the R-facts. Otherwise, if there is no S-fact among its input, it outputs nothing.

Join: $R \bowtie_\theta S$**.** As in the semijoin case, let I and J be the projection of θ to its first and second coordinates. The mapper works in the same manner as in the semijoin case. The reducer *red* works as follows. On key t, it passes through its input, while remembering all the R-facts and all its S-facts in its registers. Similar to the semijoin case, since DB is of bounded degree, say $\leq d$, the number R-facts and S-facts associated with one particular key is also bounded by d. So we can choose the number of registers in *red* to be $(k+l)d$, where k and l are the arities of R and S, respectively, to accommodate all the R-facts and S-facts. If there is at least one R-fact and one S-fact among its input, it outputs all the combination of the join among the R-facts and S-facts in the input. Otherwise, if there is no S-fact or if there is no R-fact, it outputs nothing.

D.3 Proof of Part (2) of Theorem 20

Note that for union, intersection, difference, selection and projection, one-round DSA presented above works for arbitrary database. Hence, to prove part (2) of Theorem 20, it is sufficient to show that semijoin operation $R \ltimes_\theta S$ over arbitrary graph can be done in one-round DST.

Let I and J be the projection of θ to its first and second coordinates. The mapper works similarly as follows.

$$map(t) \quad = \quad \begin{cases} \langle T(\pi_I(\bar{a})) : R(\bar{a}) \rangle & \text{if } t \text{ is } R(\bar{a}) \\ \langle T(\pi_J(\bar{a})) : S(\bar{a}) \rangle & \text{if } t \text{ is } S(\bar{a}) \\ \emptyset & \text{otherwise} \end{cases}$$

The reducer *red* works as follows. On key t, in the first pass it checks whether there is an S-fact among the input, which can be done trivially by a finite state automaton. If there is an S-fact, in the second pass on each R-fact $R(\bar{a})$ in the input, it outputs $T(\bar{a})$. If there is no S-fact, in the second pass it does nothing and output nothing.

D.4 Proof of Part (3) of Theorem 20

Again, since all the other operations can be evaluated by DSA and DST, It is sufficient to show that join operation $R \bowtie_\theta S$ over arbitrary database can be done in one-round DSTJ.

Let n_1 and n_2 be the arities of R and S, respectively. Let I and J be the projection of θ to its first and second coordinates. The mapper works similarly as follows.

$$map(t) \quad = \quad \begin{cases} \langle T(\pi_I(\bar{a})) : R(\pi_{[n_1]-I}\bar{a})) \rangle & \text{if } t \text{ is } R(\bar{a}) \\ \langle T(\pi_J(\bar{a})) : S(\pi_{[n_2]-J}\bar{a})) \rangle & \text{if } t \text{ is } S(\bar{a}) \\ \emptyset & \text{otherwise} \end{cases}$$

The reducer *red* uses joiner on each key $T(\bar{b})$ to get the On key $T(\bar{a})$, it uses joiner that outputs the bag $\{T(\bar{a}\bar{b}\bar{c}) \mid (R, \bar{b}) \in \mathsf{VAL} \text{ and } (S, \bar{c}) \in \mathsf{VAL}\}$.