Distributed Streaming with Finite Memory

Frank Neven[1], Nicole Schweikardt[2], Frédéric 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.2015.324

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 [31] and Spark [25], which support this paradigm, as well as by many specialised systems built on top of them such as Hive [29], Pig [15], Shark [32], 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 [24, 23, 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 Martín Ugarte; pp. 324–341

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 [25, 33]. 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 [22]. 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 [26, 25, 33]. By definition, DSTJs hence can perform join operations. We will show later that DSTJs can evaluate the whole class of relational algebra queries.

---

[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.

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 [30], 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, 27, 28]. 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, 27], 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.

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.

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

See, for example, the algorithm for enumerating the triangles in [27] 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 $\mathbf{K} \times \mathbf{V}$;

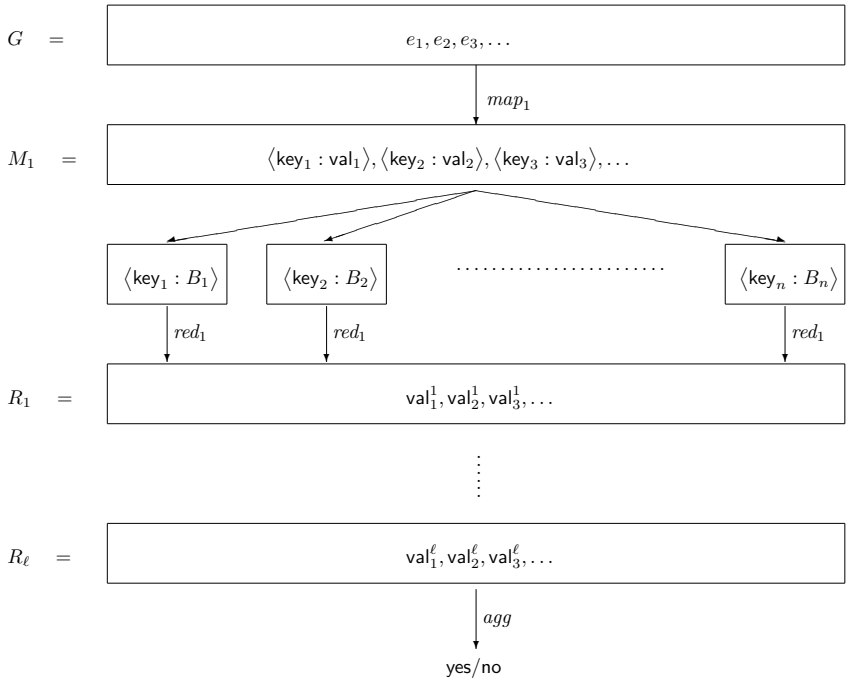


Figure 1 The flow of computation in an ℓ round key-value paradigm computation.

- 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 $\mathrm{Bags}(S \times \mathbf{D}_\#^n)$ 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, 27]. 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 [23, 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 [31] 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, 22]. 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 [22], 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; 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.

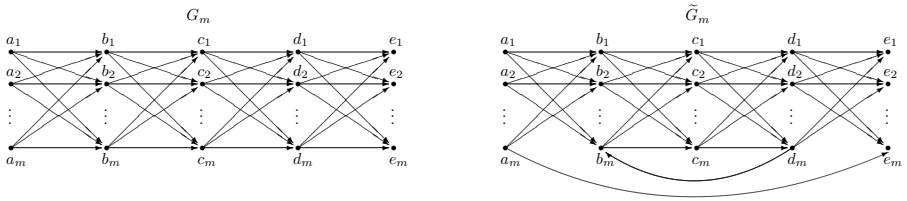


Figure 2 DSAs cannot differentiate between G_m on the left and $\widetilde{G}_m$ on the right.

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 [27]. ◀

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.

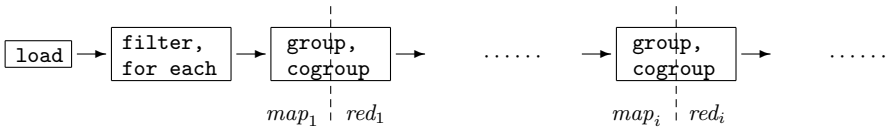
Let I and J be the projection of θ to its first and second coordinates. The mapper works as in the case of semijoin above. The reducer *red* works as follows. On key t, in the first pass it checks whether there is an S-fact among the input, which can be done trivially by a finite state automaton. If 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, 23]. 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 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.

23 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.

24 Apache Pig. Pig. `http://pig.apache.org/`.

25 Apache Spark. Spark. `http://spark.apache.org`.

26 Apache Spark. Spark programming guide. `http://spark.apache.org/docs/latest/programming-guide.html`.

27 S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *WWW*, 2011.

28 Y. Tao, W. Lin, and X. Xiao. Minimal mapreduce algorithms. In *SIGMOD*, 2013.

29 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.

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

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

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

33 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.