Verifying Equivalence of Spark Programs

Shelly Grossman[1], Sara Cohen[2], Shachar Itzhaky[3], Noam Rinetzky[1], and Mooly Sagiv[1]

[1] Tel Aviv University, Israel. {shellygr,maon,msagiv}@tau.ac.il
[2] The Hebrew University of Jerusalem, Israel. sara@cs.huji.ac.il
[3] Massachusetts Institute of Technology, USA. shachari@mit.edu

Abstract. *Spark* is a popular framework for writing large scale data processing applications. Such frameworks, intended for data-intensive operations, share many similarities with database systems, but do not enjoy a similar support of optimization tools. Our goal is to enable such optimizations by first providing the necessary theoretical setting for verifying the equivalence of Spark programs. This is challenging because Spark programs combine relational algebraic operations and aggregate operations with *User Defined Functions* (*UDF*s).
We present the first technique for verifying the equivalence of Spark programs. We model Spark as a programming language whose semantics imitates Relational Algebra queries (with aggregations) over bags (multisets) and allows for UDFs expressible in Presburger Arithmetics. We present a technique for verifying the equivalence of an interesting class of Spark programs, and show that it is complete under certain restrictions. We implemented our technique in a prototype tool, and used it to verify the equivalence of a few small, but intricate, open-source Spark programs.

1 Introduction

Spark [19, 27, 28] is a popular framework for writing large scale data processing applications. Such frameworks, intended for data-intensive operations, share many similarities with NoSQL database systems (see, e.g., [3]). Unlike traditional relational databases, which are accessed using a standard query language, NoSQL databases are often accessed via an entire program. As NoSQL databases are typically huge, optimizing such programs is an important problem. Unfortunately, although query optimization has been studied extensively over the last few decades (see Section 6), the techniques presented in the past no longer carry over when access to the data is via a rich programming language.

Standard query optimization techniques (see, e.g., [5]) are often based on the notion of query equivalence, i.e., the ability to determine if different queries are guaranteed to return the same results over all database inputs. If one can decide equivalence between queries, it may be possible to devise a procedure that simplifies a given query to derive a cheaper, yet equivalent, form. In addition, the ability to decide query equivalence is a necessary component to allow rewriting of queries with views or previously computed queries. This, again, is an important optimization technique.

This paper studies the equivalence problem in the context of Spark. Spark was developed in reaction to the data flow limitations of the Map-Reduce paradigm. Importantly, Spark programs are centered on the resilient distributed dataset (RDD) structure, which contains a bag of (distributed) items. An RDD r can be accessed using operations such as *map*, which applies a function to all items in r, *filter*, which filters items in r using a given boolean function, and *fold* which aggregates items together, again using a user defined function (UDF). Intuitively, map, filter and fold can be seen as extensions to the standard database operations *project*, *select* and *aggregation*, respectively, with arbitrary UDFs applied. A language such as *Scala* or *Python* is used as Spark's interface, which allows to embed calls to the underlying framework in standard programs, as well as to define the UDFs that Spark executes.

Unsurprisingly, in the general case, the problem of determining equivalence between Spark programs is undecidable. Thus, this paper focuses on fragments of Spark programs, and gives a sound condition for equivalence. For special cases, we also provide conditions that are complete for determining equivalence.

The techniques used in this paper for determining equivalence of Spark programs are rather different than traditional query equivalence characterizations. Query equivalence is usually determined by either *(1)* showing that equivalence boils down to the existence of some special mapping (e.g., isomorphism, homomorphism) between the queries *or (2)* proving that equivalence over arbitrary databases can be determined by checking for equivalence over some finite set of canonical databases. (Actually, often, both such characterizations are shown.) In this paper, we use a different approach. Intuitively, we present a method to translate Spark programs into a decidable theory such as Presburger arithmetic, from which we derive a decision procedure for verifying equivalence of Spark programs. We note that some of the intricacies arise from the fact RDDs are bags (and not individual items), and Spark programs can contain aggregation (using the fold operation). We also note that in this paper, we ignore the distribution aspect of Spark programs and analyze them at a higher abstraction level of bag-manipulating programs.

1.1 Overview

We extract the essence of the general Spark framework using a simple language called SparkLite, which is a functional programming language with a built-in parametric *RDD* data type and operations on it. Programs are written using sequential composition of atomic commands, *RDD* operations and conditions, but no loops. We define methods for proving equivalence of programs for the different classes of SparkLite, listed in Figure 1. We continue by example, and apply our technique to two key classes: $NoAgg$ and Agg^1.

$NoAgg$ Example. Below are two equivalent programs working on integer RDDs. Both compute a new RDD containing all the elements of the input RDD which are at least 50, multiplied by two. The programs operate differently: $P1$ first multiplies, then filters, while $P2$ goes the other way around.

Class	**Description**	**Method coverage**
$NoAgg$	No aggregations	Sound and complete
Agg^1	Single aggregation, primitive output	Sound
$AggPair^1_{sync}$	Distributive aggregations	Sound and complete
Agg^1_R	Single aggregation, RDD output	Sound
Agg^n	Multiple non-nested aggregations	Sound

Fig. 1. Classes of Spark programs and the coverage (soundness/completeness) of their equivalence verification techniques.

```
  P1($R$: $RDD_{\texttt{Int}}$):
1 $R'_1 = \texttt{map}(\lambda x.2 * x)(R)$
2 $R''_1 = \texttt{filter}(\lambda x.x \geq 100)(R'_1)$
3 return $R''_1$
```

```
P2($R$: $RDD_{\texttt{Int}}$):
$R'_2 = \texttt{filter}(\lambda x.x \geq 50)(R)$
$R''_2 = \texttt{map}(\lambda x.2 * x)(R'_2)$
return $R''_2$
```

`map` and `filter` are operations that apply a function on each element in the RDD, and yield a new RDD. For example, let RDD R be the bag $R = \{\!\{2, 2, 103, 64\}\!\}$ (note that repetitions are allowed). R is an input of both $P1$ and $P2$. The `map` operator in the first line of $P1$ produces a new RDD, R'_1, by doubling every element of R, i.e., $R'_1 = \{\!\{4, 4, 206, 128\}\!\}$. The `filter` operator in the second line generates RDD R''_1, containing the elements of R'_1 which are at least 100, i.e., $R''_1 = \{\!\{206, 128\}\!\}$. The second program first applies the filter operator, producing an RDD R'_2 of all the elements in R which are smaller than 50, resulting in the RDD $R'_2 = \{\!\{103, 64\}\!\}$. After $P2$ applies the map operator, it ends up with an RDD R''_2 containing the same elements as R''_1. Hence, both program return the same value.

To verify that the programs are indeed equivalent, we encode them symbolically using formulae in FOL, such that the question of equivalence boils down to proving the validity of a formula. In this example, we encode $P1$ as: $ite(2 * x \geq 100, 2 * x, \bot)$, and $P2$ as: $(\lambda x.2 * x)(ite(x \geq 50, x, \bot))$, where ite denotes the if-then-else operator. The variable symbol x can be thought of as an arbitrary element in the dataset R, and the terms on the left and right side of the equality sign record the effect of $P1$ and $P2$, respectively, on x. The constant symbol $\bot$ records the deletion of an element due to not satisfying the condition checked by the `filter` operation. The formula whose validity attests for the equivalence of $P1$ and $P2$ is thus: $\forall x.ite(2 * x \geq 100, 2 * x, \bot) = (\lambda x.2*x)(ite(x \geq 50, x, \bot))$. Here, the formula is expressed in a decidable extension of Presburger Arithmetic, which allows to handle $\bot$ (see Appendix A), thus its validity can be decided.

This example points out an important property of the *map* and *filter* operations, namely, their *locality*: they handle every element separately, with no regard to its multiplicity (the number of duplicates it has in the *RDD*) or the presence of other elements. Thus, we can symbolically represent the effect of the program on any *RDD* by encoding its effect on a single arbitrary element.

Agg^1 Example. Here, we consider programs with aggregations. Aggregations encapsulate a loop over all elements in an RDD, returning a single accumulated

result. Suppose we maintain a table of products and their prices, and we write a program that verifies no product has a price lower than some threshold, say 100\$. Alternatively, we might wish to determine that if we give a 20% discount for all products whose prices are above 125\$, and no discount otherwise, then no product will carry a price tag lower than 100\$. We can verify that our discount scheme, implemented using the function e.g., using $discount((prod, p)) =$ $\textbf{if}\ p \geq 125\ \textbf{then}\ (prod, 0.8p)\ \textbf{else}\ (prod, p)$, indeed preserves the threshold, by showing that programs $P3$ and $P4$, shown below, are equivalent, i.e., they return the same answer for any input RDD.

The input RDD for $P3$ and $P4$ is a comprised of pairs integers, the first element in the pair encodes the product id, and the second element its price. $P3$ assigns the minimal price to $minP$ and then returns *true* if $minP \geq 100$ and *false* otherwise. $P3$ computes the minimal price using the `fold` aggregation operation. The latter picks some arbitrary element from R and compares it to $+\infty$, the initial value, and stores the lower value in the accumulator. It then iterates over the other elements of R in an unspecified order, comparing their value to that of the accumulator, and storing in it the lower value. $P4$ first applies the *discount* function specified earlier to every element, resulting in a temporary $RDD\ R'$, and then uses `fold` to calculate the minimal price after applying the discount and returns whether it is at least 100.

Let: $min2 = \lambda A, (x, y).min(A, y)$

```
P3(R: RDD_{Prod×Int}):
$minP = \texttt{fold}(+\infty, min2)(R)$
return $minP \geq 100$
```

```
P4(R: RDD_{Prod×Int}):
$R' = \texttt{map}(\lambda(prod, p).discount((prod, p)))(R)$
$minDiscountP = \texttt{fold}(+\infty, min2)(R')$
return $minDiscountP \geq 100$
```

Program equivalence can be written in our encoding as formulas as follows:

$$[(prod, p)]_{+\infty, \lambda A, (x,y).min(A,y)} \geq 100 \iff [discount((prod, p))]_{+\infty, \lambda A, (x,y).min(A,y)} \geq 100$$

To prove it, we apply induction. In the induction base, we check for empty RDDs, for which both programs return true. Otherwise, we assume that after n prices checked, the minimum M in $P3$, and the minimum M' in $P4$, are simultaneously at least 100 or not. The programs are equivalent if this invariant is kept after checking the next price. The formula for this invariant is:

$$\begin{aligned} &\forall (prod, p), M, M'.(M \geq 100 \iff M' \geq 100) \implies \\ &\quad (min(M, p) \geq 100 \iff min(M', p_2(discount((prod, p)))) \geq 100) \end{aligned}$$

1.2 Main Results

The main contributions of this paper are as follows:

- We present a simplified model of Spark by defining a programming language called *SparkLite*, in which UDFs are expressed over a decidable theory.
- We define a sound method for checking equivalence of a broad class of SparkLite programs, which is complete in the absence of aggregations.

- We introduce an interesting and nontrivial subclass of SparkLite with aggregations in which checking program equivalence is decidable, called $AggPair^1_{sync}$. The decidability is proven by observing that SparkLite aggregates are *closed* in the sense that composed operations can be simulated by a single operation.
- An implementation of the methods in the paper using the *Z3 SMT solver* [13], and its application to real Spark code over Python.

2 The SparkLite language

In this section, we define the syntax of SparkLite, a simple functional programming language which allows to use Spark's *resilient distributed datasets* (*RDDs*) [27].

Preliminaries. We denote a (possibly empty) sequence of elements coming from a set X by $\overline{X}$. We write $ite(p, e, e')$[4] to denote an expression which evaluates to e if p holds and to e' otherwise. We use $\bot$ to denote the *undefined* value. A *bag* m over a domain X is a multiset, i.e., a set which allows for repetitions, with elements taken from X. We denote the *multiplicity* of an element x in bag m by $m(x)$, where for any x, either $0 < m(x)$ or $m(x)$ is undefined. We write $x \in m$ as a shorthand for $0 < m(x)$. We write $\{\!\!\{x; n(x) \mid x \in X \wedge \phi(x)\}\!\!\}$ to denote a bag with elements from X satisfying some property ϕ with multiplicity $n(x)$, and omit the conjunct $x \in X$ if X is clear from context. We denote the *size* (number of elements) of a set X by $|X|$ and that of a bag m of elements from X by $|m|$, i.e., $|m| = \Sigma_{x \in X}\, ite(x \in m, m(x), 0)$. We denote the empty bag by $\{\!\!\{\}\!\!\}$.

First-Order Functions	$Fdef$	$::=$ `def` $\boldsymbol{f} \ = \ \lambda\overline{\boldsymbol{y} : \boldsymbol{\tau}}.\, e : \boldsymbol{\tau}$
Second-Order Functions	$PFdef$	$::=$ `def` $\boldsymbol{F} \ = \ \lambda\overline{\boldsymbol{x} : \boldsymbol{\tau}}.\, \lambda\overline{\boldsymbol{y} : \boldsymbol{\tau}}.\, e : \boldsymbol{\tau}$
Function Expressions	f	$::= \boldsymbol{f} \mid \boldsymbol{F}(\overline{e})$
General Expressions	η	$::=$ `cartesian`$(\boldsymbol{r}, \boldsymbol{r}) \mid$ `map`$(f)(\boldsymbol{r}) \mid$ `filter`$(f)(\boldsymbol{r})$ $\mid$ `fold`$(\boldsymbol{e}, f)(\boldsymbol{r}) \mid e$
Let expressions	E	$::= Let\ \boldsymbol{x} = \eta\ in\ E \mid \eta$
Programs	$Prog$	$::= \boldsymbol{P}(\overline{\boldsymbol{r} : RDD_{\boldsymbol{\tau}}}, \overline{\boldsymbol{v} : \boldsymbol{\tau}}) \ = \ \overline{Fdef} \quad \overline{PFdef} \quad E$

Fig. 2. Syntax for SparkLite

The syntax of SparkLite is defined in Figure 2. SparkLite supports two primitive types: *integers* (`Int`) and *booleans* (`Boolean`). On top of this, the user can define *record types* $\boldsymbol{\tau}$, which are Cartesian products of primitive types, and *RDD*s: $RDD_{\boldsymbol{\tau}}$ is (the type of) bags containing elements of type $\boldsymbol{\tau}$. We refer to primitive types and tuples of primitive types as *basic types*, and, by abuse of notation, range over them using $\boldsymbol{\tau}$. We use e to denote an expression containing only basic types, without specifying the exact underlying theory.[5] We range over variables using $\boldsymbol{v}$ and $\boldsymbol{r}$ for variables of basic types and RDD, respectively.

A program $\boldsymbol{P}(\overline{\boldsymbol{r} : RDD_{\boldsymbol{\tau}}}, \overline{\boldsymbol{v} : \boldsymbol{\tau}}) \ = \ \overline{Fdef}\,\overline{PFdef}\,E$ is comprised of a *header* and a *body*, which are separated by the $=$ sign. The header contains the name

[4] ite is shorthand for if-then-else

[5] Appendix A includes an extensive discussion of such a theory.

of the program ($\boldsymbol{P}$) and the name and types of its input parameters, which may be *RDDs* ($\overline{\boldsymbol{r}}$) or records ($\overline{\boldsymbol{v}}$). The body of the program is comprised of two sequences of function declarations ($\overline{Fdef}$ and $\overline{PFdef}$) and the program's *main expression* (E). $\overline{Fdef}$ binds function names $\boldsymbol{f}$ with first-order lambda expressions, i.e., to a function which takes as input a sequence of arguments of basic types and return a value of a basic type. $\overline{PFdef}$ associates function names $\boldsymbol{F}$ with a restricted form of second-order lambda expressions, which we refer to as *parametric functions*.[6] A parametric function $\boldsymbol{F}$ receives a sequence of basic expressions and returns a first order function. Parametric functions can be instantiated to form an unbounded number of functions from a single pattern. For example, `def addC` $= \lambda x\colon$ `Int`. $\lambda y\colon$ `Int`. $x + y\colon$ `Int` allows to create any first order function which adds a constant to its argument.

The program's main expression is comprised of a sequence of *let* expression which bind general expressions to variables. A general expression is either a basic expression (e), or an RDD *expression*. The expression $\texttt{cartesian}(\boldsymbol{r}, \boldsymbol{r'})$ returns the cartesian product of *RDDs* $\boldsymbol{r}$ and $\boldsymbol{r'}$. The expressions `map` and `filter` generalize the *project* and *select* operators in *Relational Algebra* (*RA*) [1, 8], with *user-defined functions* (*UDFs*): $\texttt{map}(f)(\boldsymbol{r})$ evaluates to an *RDD* obtained by applying the UDF f to every element x of *RDD* $\boldsymbol{r}$. $\texttt{filter}(f)(\boldsymbol{r})$ evaluates to a copy of $\boldsymbol{r}$, except that all elements in $\boldsymbol{r}$ which do not satisfy f are removed. The expression `fold` is a generalization of aggregate operations in SQL, e.g., `SUM` or `AVERAGE`, with *UDFs*: $\texttt{fold}(\boldsymbol{e}, f)(\boldsymbol{r})$ accumulates the results obtained by iteratively applying f to every element x in $\boldsymbol{r}$, starting from the *initial element* e and applying f for every $x \in \boldsymbol{r}$. If $\boldsymbol{r}$ is empty, then $\texttt{fold}(\boldsymbol{e}, f)(\boldsymbol{r}) = \boldsymbol{e}$

Remarks. First, as is common in functional languages, variables are never reassigned. We assume that the signature of UDFs given to either *map*, *filter*, or *fold* match to the type of the RDD on which they are applied. Also, to ensure that the meaning of $\texttt{fold}(\boldsymbol{e}, f)(\boldsymbol{r})$ is well defined, we require that f be a commutative function, in the following sense: $\forall x, y_1, y_2 . f(f(x, y_1), y_2) = f(f(x, y_2), y_1)$. Throughout this paper, we shall use syntactic sugar for *let* expressions in examples, and write programs as a series of variable assignments.

3 Term Semantics for SparkLite

In this section, we define semantics for SparkLite where the program is interpreted as a term in FOL. This term is called the *program term* and denoted $\Phi(P)$ for program P, specified in Figure 3. Special variables are used to refer to elements of input RDDs,[7] and a new language construct is added to denote the fold operation.

[6] Parametric functions were inspired by the *Kappa Calculus* [18], which contains only first-order functions, but allows lifting them to larger product types, which is exactly the purpose of parametric functions in SparkLite.

[7] To avoid overhead of notations, we assume the programs do not contain self-products (for every cartesian product in the program, the sets of variables appearing in each component must be disjoint).

$$\begin{aligned}
&\phi_P(\mathit{Let}\ x = \eta\ \mathit{in}\ E) &&= \phi_P(E)[\phi_P(\eta)/x] \\
&\phi_P(e) &&= e \\
&\phi_P(\texttt{map}(f)(r)) &&= f(\phi_P(r)) \\
&\phi_P(\texttt{filter}(f)(r)) &&= \mathit{ite}(f(\phi_P(r)) = \mathit{tt}, \phi_P(r), \bot) \\
&\phi_P(\texttt{cartesian}(r_1, r_2)) &&= (\phi_P(r_1), \phi_P(r_2)) \\
&\phi_P(\texttt{fold}(f, e)(r)) &&= [\phi_P(r)]_{e,f} \\
&\phi_P(r) &&= \begin{cases} \mathbf{x}_r & r \in \overline{r} \\ r & \mathit{otherwise} \end{cases}
\end{aligned}$$

$$\mathit{Let}\ P : \boldsymbol{P}(\overline{\boldsymbol{r}}, \overline{\boldsymbol{v}}) = \overline{\boldsymbol{F}}\,\overline{\boldsymbol{f}}\,E$$

$$\varPhi(P) = \phi_P(E)$$

Fig. 3. Compiling SparkLite to Logical Terms

The variables assigned by $\varPhi$ for input RDDs are called *representative elements.* In a program that receives an input RDD r, we denote the representative element of r as $\mathbf{x}_r$. The set of possible valuations to that variable is equal to the bag defined by r, and an additional 'undefined' value ($\bot$), for the empty RDD. Therefore $\mathbf{x}_r$ ranges over $\mathrm{dom}(r) \cup \{\bot\}$. By abuse of notations, the term for a non-input RDD, computed in a SparkLite program, is also called a representative element.

In Figure 3, we specify how programs written in SparkLite are translated to logical terms. Our programs are written as a series of 'let' expressions of the form $\mathit{Let}\ x = \eta\ \mathit{in}\ E$. The translation replaces all instances of x in E with the term for η, which is computed by a recursive call to ϕ_P. Input RDDs are simply translated to their representative element variable. If ϕ_P is applied on an RDD which is not an input RDD, it is not modified. By construction, it is guaranteed that this can only happen in the context of a 'let' expression, so after full evaluation of the 'let' expression, the resulting term has only variables which are representative elements of input RDDs. Non-RDD expressions are not modified by ϕ_P. Map is translated to an application of the map UDF f on the term of the RDD on which we apply the map. Filter is translated to an *ite* expression, which returns the term of the RDD on which the `filter` operator is applied, if that term satisfies the given UDF f, and no element otherwise (represented by the value $\bot$). The cartesian product is translated to a tuple of terms of the argument RDDs. `fold` is challenging, as it cannot be expressed as a first-order term. We solve this by introducing a special notation for the folded value of a term of an RDD r with a given initial value e and fold UDF f: $[\phi_P(r)]_{e,f}$.

Semantics of SparkLite in FOL form. Let P be a SparkLite program. We use ρ_0 to denote the environment of input variables and function definitions. The semantics of a program that returns an RDD-type output is the bag that is obtained from all possible valuations to the free variables: $[\![\varPhi(P)]\!](\rho_0)$, where $[\![\varPhi(P)]\!]$ is defined in Figure 4. Assigning a concrete valuation to the free variables of $\varPhi(P)$ returns an element in the output RDD r^{out}. By taking all possible valuations to the term with elements from the input RDDs $\bar{r}$, we get the bag equal to r^{out}. For fold operations, a non-RDD value is returned.

$$
\begin{aligned}
\llbracket \mathbf{x}_{r_i} \rrbracket(\rho_0) &= \rho_0(r_i)\\
\llbracket v \rrbracket(\rho_0) &= \rho_0(v)\\
\llbracket f \rrbracket(\rho_0) &= \rho_0(f)\\
\llbracket f(t_1,\dots,t_n) \rrbracket(\rho_0) &= \{\!\{ \llbracket f \rrbracket(y_1,\dots,y_n) \mid \mathit{if}\ t_i \in \mathit{RDD}\ \mathit{then}\ y_i \in \llbracket t_i \rrbracket(\rho_0)\ \mathit{else}\ y_i = \llbracket t_i \rrbracket(\rho_0) \}\!\}\\
\llbracket \mathit{ite}(f(t), t, \bot) \rrbracket(\rho_0) &= \{\!\{ z \mid z \in \llbracket t \rrbracket(\rho_0) \wedge \llbracket f \rrbracket(\rho_0)(z) \}\!\}\\
\llbracket (t, t') \rrbracket(\rho_0) &= \{\!\{ (z, z') \mid z \in \llbracket t \rrbracket(\rho_0) \wedge z' \in \llbracket t \rrbracket(\rho_0) \}\!\}\\
\llbracket [t]_{e,f} \rrbracket(\rho_0) &= q_f\big(\llbracket e \rrbracket(\rho), \llbracket t \rrbracket(\rho_0)\big)\\
& \quad q_f(v_0, s) = \begin{cases} v_0 & s = \{\!\{\}\!\} \\ \rho(f)\big(q_f(v_0, s'), x\big) & s = \{\!\{x; 1\}\!\} \cup s' \end{cases}
\end{aligned}
$$

Fig. 4. Semantics of Terms

4 Verifying Equivalence of SparkLite Programs

In this section we present techniques for verifying the equivalence of SparkLite programs, based on the representation of programs as logical terms. We begin by formally defining the program equivalence problem.

The Program Equivalence (*PE*) problem. Let P_1 and P_2 be SparkLite programs, with signature $P_i(\overline{T}, \overline{RDD_T})\colon \tau$ for $i \in \{1, 2\}$. We denote by ρ_0 the environment of input variables $\overline{v}$, input RDDs $\overline{r}$ and function definitions. We use $\llbracket P_i \rrbracket(\rho_0)$ to denote the result of P_i. We say that P_1 and P_2 are *equivalent*, if for all environments ρ_0, it holds that $\llbracket P_1 \rrbracket(\rho_0) = \llbracket P_2 \rrbracket(\rho_0)$.

4.1 Verifying Equivalence of SparkLite Programs w.o. Aggregations

We are given two programs P_1, P_2 receiving as input a series of RDDs $\overline{r} = (r_1, \dots, r_n)$. We assume w.l.o.g. the programs receive only RDD arguments $\overline{r}$.[8] We let $\overline{x} = (x_1, \dots, x_n)$ be a concrete valuation for all input representative elements: $\mathbf{x}_{r_i}$ will map to x_i. We denote the substitution of the concrete valuation in a term t over $\overline{\mathbf{x}_r}$ as: $t(\overline{x}) = t[x_1/\mathbf{x}_{r_1}, \dots, x_n/\mathbf{x}_{r_n}]$.

We define a class of programs without aggregations in the program term.

Definition 1 (The *NoAgg* class). *A program P satisfies $P \in NoAgg$ if $\Phi(P)$ does not contain any aggregate terms (i.e. terms of the form: $[t]_{i,f}$).*

In Figure 5 we present an algorithm for deciding the equivalence of *NoAgg* programs. The algorithm first checks if both programs return an empty RDD for all inputs, in which case they are trivially equivalent. Otherwise, we note that it is not enough to compare the evaluations of program terms, but also the multiplicities of those values in the output RDD, as the below example illustrates.

Example 1 (The importance of multiplicities for equivalence). Let $P1(R_0, R_1) = \texttt{map}(\lambda x.1)(R_0)$ and $P2(R_0, R_1) = \texttt{map}(\lambda x.1)(R_1)$. $P1$ and $P2$ have the same program term (the constant 1),[9] but the multiplicity of that element in the output

[8] The extension to equivalence of terms which is based also on non-RDD inputs is immediate by quantification on the non-RDD variables in the term.

[9] To be more precise, the term is an application of the function $\lambda x.1$ on a free variable, $\mathbf{x}_{R_0}$ or $\mathbf{x}_{R_1}$. ϕ does not apply beta-reduction of the UDFs.

```
if $\Phi(P_1) = \bot \wedge \Phi(P_2) = \bot$ then
    return equivalent
else
    if $FV(\Phi(P_1)) \neq FV(\Phi(P_2))$ then
        return not equivalent
    else
        if $\exists \overline{x}.\Phi(P_1)[\overline{x}/FV(\Phi(P_1))] \neq \Phi(P_2)[\overline{x}/FV(\Phi(P_2))]$ then
            return not equivalent
        else
            return equivalent
```

Fig. 5. Algorithm for Deciding $NoAgg$

bag is different and depends on the source input RDD. In $P1$, its multiplicity is the same as the size of R_0, and in $P2$ it is the same as the size of R_1. $P1$ and $P2$ are thus not equivalent for inputs R_0, R_1 of different sizes. Therefore, for each program term $\Phi(P)$ we consider the *set of free variables*, denoted $FV(\Phi(P))$. Each free variable is associated with an input RDD. In the example, $FV(\Phi(P1)) = \{\mathbf{x}_{R_0}\}$, and $FV(\Phi(P2)) = \{\mathbf{x}_{R_1}\}$.

This example provides the motivation for defining sets of free variables, and for the algorithm to compare these sets. If the sets of free variables are found to be equal, the algorithm solves the equivalence formula of the program terms.

Non-empty RDDs can be equal only if the sets of free variables of the terms are equal. Otherwise, even if the terms themselves evaluate to the same element for some valuation, the multiplicity of these elements is different in the two resulting RDDs, as the underlying variables have different multiplicities in their input RDDs. We show that in two programs without aggregations, different sets of free variables of the terms imply the existence of input RDDs for which the two program terms evaluate to different bags, thus they are semantically inequivalent.

Proposition 1. *Let there be two programs $P_1, P_2 \in NoAgg$, over input RDDs $\overline{r}$ and program terms $\Phi(P_i) = t_i$. such that $FV(t_1) \neq FV(t_2)$, and $t_1 \neq \bot \vee t_2 \neq \bot$. Then, $\exists \overline{r}.[\![t_1]\!](\overline{r}) \neq [\![t_2]\!](\overline{r})$.*

Last, we note that the underlying theory of our terms should be a satisfiable one. Appendix A includes a description of an extension to Presburger arithmetic which can serve as the underlying theory of SparkLite's basic expressions, and is also decidable. We omitted the formal discussion due to considerations of space.

Theorem 1 (Decidability of the $NoAgg$ class). *Given two* SparkLite *programs $P_1, P_2 \in NoAgg$,* PE *is decidable.*

An example illustrating the theorem can be found in the overview section.

4.2 Verifying Equivalence of SparkLite Programs with Aggregation

In this section, we discuss how the existing framework can be extended to prove equivalence of SparkLite programs containing aggregate expressions. The terms for aggregate operations are given using a special operator $[t]_{i,f}$, where t is the term being folded, i is the initial value, and f is the fold function. The operator *binds* all free variables in the term t, thus the free variables of t are not contained in the free variables set of the term that includes the aggregate term.

As *PE* for general SparkLite programs is undecidable, we consider classes of SparkLite.

Theorem 2. *Hilbert's 10^{th} problem reduces to* PE.

Corollary 1. PE *is undecidable.*

Single Aggregate The simplest class of programs in which an aggregation operator appears, is the class of programs whose program terms are a function of an aggregate term, that is have the form $g([t]_{i,f})$.

Definition 2 (The Agg^1 class). *Let there be a program P with $\Phi(P) = g([t]_{i,f})$. $P \in Agg^1_R$ if t does not contain aggregate terms.*

The most simple case in which two Agg^1 programs are equivalent, is when the fold function f does not change the initial value i:

Definition 3 (Trivial fold). $[\varphi]_{i,f}$ *is a* trivial fold *if* $\forall \bar{v}. f(i, \varphi(\bar{v})) = i$

If two instances of Agg^1: $\Phi(P1) = g_1([\varphi_1]_{i_1,f_1}), \Phi(P2) = g_2([\varphi_2]_{i_2,f_2})$ have trivial folds, then equality of the initial values under g (i.e. $g_1(i_1) = g_2(i_2)$) is enough to show equivalence:

$$g_1([\varphi_1]_{i_1,f_1}) = g_1(i_1) \wedge g_2([\varphi_2]_{i_2,f_2}) = g_2(i_2) \wedge g_1(i_1) = g_2(i_2) \implies g_1([\varphi_1]_{i_1,f_1}) = g_2([\varphi_2]_{i_2,f_2})$$

When the *fold* is not trivial, we require the sets of free variables to be equal. We saw that equal sets of free variables imply equally sized RDDs. This allows us to formulate the equivalence of the *fold* operation as an inductive process: By first checking the *fold* results are equal for empty RDDs, and then for every element produced by an assignment of the free variables. The sequence of assignments must be of equal size for both RDDs, as the sets of free variables are equal, so the inductive computation terminates at the same time in both *fold* operations. While *fold* operations may be equal even on RDDs of different size, we wish to avoid such peculiarities, and require equal sets of free variables in aggregate terms to be able to apply the inductive technique for equivalence verification.

Remark. If f is used as a fold function, then $\forall A. f(A, \perp) = A$. The motivation is to avoid update of the aggregated value when f is applied on elements that were filtered out from the RDD previously.

The following lemma presents a method for proving equivalence of Agg^1 programs based on induction:

Lemma 1 (Sound method for verifying equivalence of Agg^1 programs). *Let P_1, P_2 be Agg^1 programs, with $\Phi(P_i) = g_i([\varphi_i]_{i_i,f_i})$. The terms φ_1, φ_2 are representative elements of RDDs R_1, R_2 of types σ_1, σ_2, respectively. Let f_1 :*

$\xi_1 \times \sigma_1 \to \xi_1, f_2 : \xi_2 \times \sigma_2 \to \xi_2$ *b* fold *functions,* $i_1 \colon \xi_1, i_2 \colon \xi_2$ *be initial values, and* $g_1 : \xi_1 \to \xi, g_2 : \xi_2 \to \xi$ *functions. We have* $g_1([\varphi_1]_{i_1,f_1}) = g_2([\varphi_2]_{init_2,f_2})$ *if:*

$$FV(\varphi_1) = FV(\varphi_2) \tag{1}$$

$$g_1(i_1) = g_2(i_2) \tag{2}$$

$$\forall \bar{v}, A_{\varphi_1} : \xi_1, A_{\varphi_2} : \xi_2 . g_1(A_{\varphi_1}) = g_2(A_{\varphi_2}) \implies g_1(f_1(A_{\varphi_1}, \varphi_1(\bar{v}))) = g_2(f_2(A_{\varphi_2}, \varphi_2(\bar{v}))) \tag{3}$$

An example illustrating of the lemma can be found in the overview section.

The application of Lemma 1 to the algorithm presented in Theorem 1 involves one syntactic check (Equation (1)), and two calls to the solver (Equations (2) and (3)). Lemma 1 shows that an inductive proof of the equality of folded values is *sound*. Therefore, given two folded expressions which are not equivalent, the lemma is guaranteed to report so.

A Complete Subclass There are several cases in which one or more of the requirements of Lemma 1 are not satisfied, yet the aggregate expressions are equal. Moreover, some of these cases can be identified and subsequently have equivalence verified. The requirement of Equation (1) ($FV(\varphi_0) = FV(\varphi_1)$), is unnecessary when both *fold* expressions are trivial (see Definition 3). We show an example which Lemma 1 does not cover due to Equation (3).

Example 2 (Non-injective modification of folded expressions). Non-injective transformations of the aggregate expression can weaken the inductive claim, sometimes rendering it incorrect. Due to this phenomenon, Lemma 1 does not prove the equivalence of the following Agg_1 programs. The two programs return a boolean value, indicating if the sum of the elements in the input RDD R is divisible by 3. The difference is that $P1$ takes the sum modulo 3 of each element modulo 3, and $P2$ applies modulo 3 on the original sum of the elements:

P1(R: $RDD_{\texttt{Int}}$):	**P2**(R: $RDD_{\texttt{Int}}$):
1 $R' = \texttt{map}(\lambda x.x\%3)(R)$	$v = \texttt{fold}(0, \lambda A, x.A + x)(R)$
2 $\texttt{return fold}(0, \lambda A, x.(A + x)\%3)(R') = 0$	$\texttt{return } v\%3 = 0$

The equivalence formula is: $[x\%3]_{0,+\%3} = 0 \iff [x]_{0,+}\%3 = 0$. Taking $g_1(x) = \lambda x.x = 0$, $g_2(x) = \lambda x.(x\%3) = 0$, Equation (3) is:

$$\forall x, A, A'.A = 0 \iff A'\%3 = 0 \implies (A + x\%3)\%3 = 0 \iff (A' + x)\%3 = 0$$

A counterexample is found: $A = 1, A' = 2, x = 1$. The hypothesis $A = 0 \iff A'\%3 = 0$ is satisfied, but $(A + x\%3)\%3 = 2 \neq 0$, while $(A' + x)\%3 = 0$.

Despite the fact that Lemma 1 did not cover Example 2, this example belongs to a subclass of Agg^1 for which a sound and complete equivalence test method exists. This class is characterized by a verifiable semantic property of the fold functions and the initial values. This semantic property states that an application of *fold* on a sequence of two elements can be expressed as an application of *fold* on a single element. For example, if the fold function is *sum*, we say that applying *sum* on an aggregated value A and two elements x, y can be written as a sum of A and $x + y$. We name this process 'shrinking' for short. We say that two programs belong to a class called $AggPair^1_{sync}$ if for both programs, it is possible to 'shrink' a sequence of iterated applications of the fold function starting from the initial value, using the same element in both programs.

Definition 4 (The $AggPair^1_{sync}$ class). *Let there be two Agg^1 programs P_1, P_2 with equal signatures, whose program terms are $g_j([\varphi_j]_{i_j,f_j})$ for $j = 1, 2$. We say that $\langle P_1, P_2\rangle \in AggPair^1_{sync}$ if:*

$$FV(\varphi_1) = FV(\varphi_2) \tag{4}$$

$$\begin{aligned} \forall \bar{v_1}, \bar{v_2}.\exists \bar{v}'.f_1(f_1(i_1, \varphi_1(\bar{v_1})), \varphi_1(\bar{v_2})) &= f_1(i_1, \varphi_1(\bar{v}')) \\ \wedge\, f_2(f_2(i_2, \varphi_2(\bar{v_1})), \varphi_2(\bar{v_2})) &= f_2(i_2, \varphi_2(\bar{v}')) \end{aligned} \tag{5}$$

The $AggPair^1_{sync}$ class contains pairs of programs in which repeated application of Equation (5) can shrink multiple applications of the *fold* function starting from the same initial value and on the same sequence of valuations, to a single application of the *fold* function, using the same valuation for both programs.

Theorem 3 (Equivalence in $AggPair^1_{sync}$ is *decidable*). *Let P_1, P_2 such that $\langle P_1, P_2\rangle \in AggPair^1_{sync}$, with input RDDs $\bar{r}$. Let $\Phi(P_j) = g_j([\varphi_j]_{i_j,f_j})$. Then, $\Phi(P_1) = \Phi(P_2)$ if and only if:*

$$g_1(i_1) = g_2(i_2) \tag{6}$$

$$\begin{aligned} &\forall \bar{v}, \bar{y}, A_1, A_2.(A_1{=}f_1(i_1, \varphi_1(\bar{y})) \wedge A_2{=}f_2(i_2, \varphi_2(\bar{y})) \wedge g_1(A_1){=}g_2(A_2)) \\ &\quad \implies g_1(f_1(A_1, \varphi_1(\bar{v}))) = g_2(f_2(A_2, \varphi_2(\bar{v}))) \end{aligned} \tag{7}$$

Example 3 (Completing Example 2). We have:

$$f_0(f_0(0, x\%3), y\%3) = (x\%3)\%3 + (y\%3)\%3 = ((x + y)\%3)\%3 = f_0(0, (x + y)\%3)$$
$$f_1(f_1(0, x), y) = x + y = f_1(0, x + y)$$

So Equation (5) is true (for arbitrary x, y, $x + y$ can reduce the two fold applications), and the programs belong to $AggPair^1_{sync}$. We are left with proving:

$$\forall x, y.((0{+}y\%3)\%3 = 0 \iff (0{+}y)\%3{=}0) \implies ((y{+}x\%3)\%3 = 0 \iff (y{+}x)\%3 = 0)$$

which is correct — proving the equivalence with the stronger lemma.

Note that checking if two programs P_1, P_2 belong to $AggPair^1_{sync}$ involves a syntactic check of the free variables, and solving an additional formula (Equation (5)).

Sound Methods for Additional Classes A natural extension of the Agg^1 class is to programs that use an aggregated expression in another RDD operation. For example, filtering elements strictly larger than any element in another RDD: $\texttt{filter}((\lambda x.\lambda y.y > x)(\texttt{fold}(-\infty, max)(R_1)))(R_0)$. The program term is $ite(\mathbf{x}_{R_0} > [\mathbf{x}_{R_1}]_{-\infty,max}, \mathbf{x}_{R_0}, \bot)$.

Definition 5 (The Agg^1_R class). *Let there be a program P with $\Phi(P) = \psi$. We say that $P \in Agg^1_R$ if ψ contains a single instance of an aggregate term in a position p: $\psi|_p = [\varphi]_{i,f}$, which is denoted γ, and in addition, φ has no aggregate terms. We write $\Phi(P) = \psi[\gamma]|_p$, where γ the value of the aggregate sub-term.*

Lemma 2 (Lifting Lemma 1 to Agg^1_R). *Let two* SparkLite *programs P_1, P_2 in Agg^1_R with terms ψ_j and aggregate expressions $\gamma_j = [\varphi_j]_{i_j,f_j}$, $j \in \{1,2\}$ in position p_j. P_1 is equivalent to P_2 if:*

$$FV(\varphi_1) = FV(\varphi_2) \tag{8}$$
$$FV(\psi_1) = FV(\psi_2) \tag{9}$$
$$\forall \bar{x}.\psi_1[i_1]|_{p_1}(\bar{x}) = \psi_2[i_2]|_{p_2}(\bar{x}) \tag{10}$$
$$\begin{aligned}&\forall \bar{x}, \bar{v}, A_1, A_2.(\psi_1[A_1]|_{p_1}(\bar{x}) = \psi_2[A_2]|_{p_2}(\bar{x})) \implies \\ &\qquad (\psi_1[f_1(A_1, \varphi_1(\bar{v}))]|_{p_1}(\bar{x}) = \psi_2[f_2(A_2, \varphi_2(\bar{v}))]|_{p_2}(\bar{x}))\end{aligned} \tag{11}$$

Lemmas 1,2 show that Agg^1, Agg^1_R have a sound equivalence verification method, and Theorem 3 shows that $AggPair^1_{sync}$ has a sound and complete equivalence verification method.[10] The sound technique can be further generalized to programs with multiple aggregate terms, where the aggregated terms are not nested — each aggregate term does not contain an aggregate term in its definition. We denote this class Agg^n.

Definition 6 (The Agg^n class). *Let there be a program P with $\Phi(P) = g([t_1]_{i_1,f_1}, \ldots, [t_n]_{i_n,f_n})$, or $g(\overline{[t_i]_{i_i,f_i}})$ for short. $P \in Agg^n$ if $t_1, \ldots, t_n$ do not contain aggregate terms.*

Lemma 3. *Let P_1, P_2 be two programs in Agg^n, such that $\Phi(P_j) = g_j(\overline{[\varphi_j]_{i_j,f_j}})$. We have $g_1(\overline{[\varphi_1]_{i_1,f_1}}) = g_2(\overline{[\varphi_2]_{i_2,f_2}})$ if:*

$$\forall j_1, j_2.FV(\varphi_{1,j_1}) = FV(\varphi_{2,j_2}) \tag{12}$$
$$g_1(\overline{i_1}) = g_2(\overline{i_2}) \tag{13}$$
$$\forall \overline{v}, \overline{A_1}, \overline{A_2}.g_1(\overline{A_1}) = g_2(\overline{A_2}) \implies g_1(\overline{f_1(A_1, \varphi_1(\overline{v}))}) = g_2(\overline{f_2(A_2, \varphi_2(\overline{v})})) \tag{14}$$

We note that the subset of Agg^n programs that can be handled with Lemma 3 could be extended if we relaxed Equation (12). Lemma 3 requires all aggregate terms to be on RDDs of the same size (disregarding filter operations — the induction is on the number of possible valuations due to the sets of free variables), to allow equal induction lengths. We show a motivating example for relaxing Equation (12):

[10] Even when the completeness criterion for $AggPair^1_{sync}$ is not met, we may be successful in proving the equivalence using Lemma 1. For example, $[((\lambda x.1)(\mathbf{x}_{r_0}), (\lambda x.1)(\mathbf{x}_{r_1}))]_{0,\lambda A,(x,y).A+x+y} = [((\lambda x.1)(\mathbf{x}_{r_1}), (\lambda x.1)(\mathbf{x}_{r_0}))]_{0,\lambda A,(x,y).A+x+y}$ can be proved by induction, but for $f = \lambda A, (x,y).A + x + y$, $f(f(A,(1,1)),(1,1)) = A + 4 \neq f(A,(1,1)) = A + 2$ (the choice of valuation does not change the result). Thus, it does not satisfy the completeness criterion.

Example 4. Let two Agg^n programs $P1, P2$ that sum the elements of an input RDD R_0. $P2$ will also apply a trivial fold on input RDD R_1 and return the sum of the aggregations. As the fold on R_1 is trivial, it will not affect the final result.

```
**P1**($R_0 \colon RDD_{\texttt{Int}}, R_1 \colon RDD_{\texttt{Int}}$):
  $v = \texttt{fold}(0, \lambda A, x.A + x)(R_0)$
  return $v$
```

```
**P2**($R_0 \colon RDD_{\texttt{Int}}, R_1 \colon RDD_{\texttt{Int}}$):
  $v' = \texttt{fold}(0, \lambda A, x.A + x)(R_0)$
  $u = \texttt{fold}(0, \lambda A, x.0)(R_1)$
  return $v' + u$
```

We see that as $P2$ has an aggregate term with a set of free variables equal to $\{\mathbf{x}_{R_1}\}$ and the other aggregate terms have $\{\mathbf{x}_{R_0}\}$, Lemma 3 returns 'not equivalent', while $P1$ and $P2$ are actually equivalent.

We note that in order to analyze such programs, we need to verify equivalence in the case some of the participating RDDs are empty, while others are not, or in the general case where *fold* operations do not terminate at the same point. However, Agg^n contains non-trivial programs, as the below example shows:

Example 5 (Independent fold). The below programs return a tuple containing the sum of positive elements in its first element, and the sum of negative elements in the second element. With lemma 3, we are able to show the equivalence.

Let: $h : (\lambda(P, N), x.ite(x \geq 0, (P + x, N), (P, N - x))$

```
**P1**($R \colon RDD_{\texttt{Int}}$):
  return $\texttt{fold}((0,0), h)(R)$
```

```
**P2**($R \colon RDD_{\texttt{Int}}$):
  $R_P = \texttt{filter}(\lambda x.x \geq 0)(R)$
  $R_N = \texttt{map}(\lambda x. - x)(\texttt{filter}(\lambda x.x < 0)(R))$
  $p = \texttt{fold}(0, \lambda A, x.A + x)(R_P)$
  $n = -\texttt{fold}(0, \lambda A, x.A + x)(R_N)$
  return $(p, n)$
```

$$\Phi(P1) = [\mathbf{x}_R]_{(0,0),h}; \quad \Phi(P2) = ([\phi_{P2}(R_P)]_{0,+}, -[\phi_{P2}(R_N)]_{0,+})$$
$$\phi_{P2}(R_P) = ite(\mathbf{x}_R \geq 0, \mathbf{x}_R, \perp)\,; \; \phi_{P2}(R_N) = ite(\mathbf{x}_R < 0, -\mathbf{x}_R, \perp)$$

We set $g_1 = g_2 = \lambda(x, y).(x, y)$, and apply Lemma 3 to prove:

$$[\mathbf{x}_R]_{(0,0),h} = ([ite(\mathbf{x}_R \geq 0, \mathbf{x}_R, \perp)]_{0,+}, -[ite(\mathbf{x}_R < 0, -\mathbf{x}_R, \perp)]_{0,+})$$

We note that Equation (12) is satisfied: $FV(R) = FV(R_P) = FV(R_N) = \{x_R\}$. Induction base case (Equation (13)) is trivial. Induction step (Equation (14)):

$$\forall x, A, B, C.p_1(A) = B \wedge p_2(A) = C \implies$$
$$p_1(h(A, x)) = B + ite(x \geq 0, x, 0) \wedge p_2(h(A, x)) = C + ite(x < 0, -x, 0)$$

5 Prototype Implementation

We developed a prototype implementation verifying the equivalence of Spark programs. The tool accepts Spark code written in the Python interface, and verifies equivalence of functions structured like SparkLite programs. The tool is written in Python 2.7 and uses the Z3 Python interface to prove formulas. A total of 22 test-cases of both equivalent and non-equivalent instances were tested, including examples from this paper. The examples were inspired by real Spark uses taken from [19,26] and online resources (e.g., open-source Spark clients). The examples include join optimizations, different aggregations, aggregation by key and various UDFs (including uninterpreted). The main technical challenge is the symbolic analysis of UDFs, given as pure Python code. All examples were verified in less than 0.2 seconds on a 64-bit Ubuntu host with a quad core 2.30 GHz Intel Core i5-6200U processor, with 8GB memory.

6 Related Work

This paper bridges the areas of databases and programming languages. The problem considered (i.e., determining equivalence of expressions accessing a dataset) is a classic topic in database theory. The solution approach (i.e., translation into a symbolic representation in a decidable theory) is one that is often employed in the programming language community. In this section we discuss related work from both of these areas.

Query containment and equivalence were first studied in the seminal work [4]. This work was extended in numerous papers, e.g., [20] for queries with inequalities and [7] for acyclic queries. Of most relevance to this paper are the extensions to queries evaluated under bag and bag-set semantics [6], and to aggregate queries, e.g., [10, 11, 15]. The latter papers consider specific aggregate functions, such as min, count, sum and average, or aggregate functions defined by operations over abelian monoids. Equivalence is characterized in terms of special types of homomorphisms or by considering a finite set of canonical databases. Equivalence characterizations are the basis for rewriting techniques, which are important both for optimization, and for data integration. See [16, 17] for a survey of the main results for non-aggregate queries. Rewriting of aggregate queries was studied in [9, 15].

Previous results on equivalence and rewriting of aggregate queries differ from the current setting significantly, in two ways. First, previous work either considered queries with specific system-defined aggregate functions (such as min, count, sum) or viewed aggregate functions as a "black box" defined abstractly using a commutative and associative operator over some set of values. In this work, aggregate functions are user-defined, and we are given access to their definitions. Hence, equivalence depends on the actual operations of arbitrary functions. Second, previous work studied equivalence of aggregate queries with the same aggregate function (i.e., equivalence of two sum-queries or equivalence of two min-queries). This paper, on the other hand, studies the equivalence problem even when programs employ different aggregate functions. These two differences are one reason why Spark program equivalence is significantly harder to determine, and yet add scope for new and surprising program equivalences to be discovered. For these reasons also, the results in this paper are not directly comparable to previous work.

SG: PL Relatd work

7 Conclusion and Future Work

To conclude, we saw that the problem of checking query equivalence (where queries were written as programs in the SparkLite language), can be modeled with logical formulas. We showed that in the presentation of a query equivalence instance as a logical formula, the solver for the formula is capable of proving equivalence of conjunctive queries (Theorem 1). Furthermore, we provided a classification of programs with the fold operation (aggregations), and presented a sound method for equivalence testing for each presented class (Lemmas 1, 2, 3), and a sound and complete method for one particular non-trivial class of programs (Theorem 3).

We hope the foundations laid in this paper will lead to development of tools that handle formal verification and optimization of clients written in Spark and similar frameworks, by building upon the concepts presented here to more elaborate structures, i.e. queries with nested aggregation, unions, and multiple step-inductions for self joins.

References

1. Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995.
2. Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.
3. Rick Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.
4. Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, STOC '77, pages 77–90, New York, NY, USA, 1977. ACM.
5. Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 34–43, 1998.
6. Surajit Chaudhuri and Moshe Y. Vardi. Optimization of real conjunctive queries. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '93, pages 59–70, New York, NY, USA, 1993. ACM.
7. Chandra Chekuri and Anand Rajaraman. Conjunctive query containment revisited. *Theoretical Computer Science*, 239(2):211 – 229, 2000.
8. E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.
9. Sara Cohen, Werner Nutt, and Yehoshua Sagiv. Rewriting queries with arbitrary aggregation functions using views. *ACM Trans. Database Syst.*, 31(2):672–715, June 2006.
10. Sara Cohen, Werner Nutt, and Yehoshua Sagiv. Deciding equivalences among conjunctive aggregate queries. *J. ACM*, 54(2), 2007.
11. Sara Cohen, Yehoshua Sagiv, and Werner Nutt. Equivalences among aggregate queries with negation. *ACM Trans. Comput. Logic*, 6(2):328–360, April 2005.
12. David C Cooper. Theorem proving in arithmetic without multiplication. *Machine Intelligence*, 1972.
13. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1792734.1792766.
14. Michael J. Fischer and Michael O. Rabin. Super-exponential complexity of presburger arithmetic. Technical report, Massachusetts Institue of Technology, Cambridge, MA, USA, 1974.
15. Stéphane Grumbach, Maurizio Rafanelli, and Leonardo Tininini. On the equivalence and rewriting of aggregate queries. *Acta Inf.*, 40(8):529–584, 2004.
16. Ashish Gupta and Iderpal Singh Mumick, editors. *Materialized Views: Techniques, Implementations, and Applications.* MIT Press, Cambridge, MA, USA, 1999.
17. Y. Alon Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.
18. Masahito Hasegawa. *Decomposing typed lambda calculus into a couple of categorical programming languages*, pages 200–219. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995.

19. Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning Spark: Lightning-Fast Big Data Analytics*. O'Reilly Media, Inc., 1st edition, 2015.
20. Anthony Klug. On conjunctive queries containing inequalities. *J. ACM*, 35(1):146–160, January 1988.
21. Viktor Kuncak, Huu Hai Nguyen, and Martin C. Rinard. Deciding boolean algebra with presburger arithmetic. *J. Autom. Reasoning*, 36(3):213–239, 2006.
22. Aless Lasaruk and Thomas Sturm. *Effective Quantifier Elimination for Presburger Arithmetic with Infinity*, pages 195–212. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
23. Derek C. Oppen. A 222pn upper bound on the complexity of presburger arithmetic. *Journal of Computer and System Sciences*, 16(3):323 – 332, 1978.
24. M. Presburger. ÃIJber die vollstÃďndigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervor. *Comptes Rendus du I congrÃÍs de MathÃľmaticiens des Pays Slaves*, pages 92–101, 1929.
25. Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning*, volume 1. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 2001.
26. Josh Wills, Sean Owen, Uri Laserson, and Sandy Ryza. *Advanced Analytics with Spark: Patterns for Learning from Data at Scale*. O'Reilly Media, Inc., 1st edition, 2015.
27. Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX.
28. Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

A A decidable extension of Presburger Arithmetic suitable for SparkLite

Presburger Arithmetic. We consider a fragment of first-order logic (FOL) with equality over the integers, where expressions are written in the rather standard syntax specified in Figure 6.[11] Disregarding the tuple expressions ($(pe\ , \overline{pe})$ and $\boldsymbol{p}_i(e)$) and ite, the resulting first-order theory with the usual $\forall$ and $\exists$ quantifiers is called the *Presburger Arithmetic.* The problem of checking whether a sentence in Presburger arithmetic is valid has long been known to be decidable [14, 24], even when combined with Boolean logic [2, 21],[12], and infinities [22].[13] For example, *Cooper's Algorithm* [12] is a standard decision procedure for Presburger Arithmetic[14].

In this paper, we consider a simple extension to this language by adding a *tuple constructor* $(pe\ , \overline{pe})$, which allows us to create k-tuples, for some $k \geq 1$, of primitive expressions, and a projection operator $\boldsymbol{p}_i(e)$, which returns the i-th component of a given tuple expression e. We extend the equality predicate to tuples in a point-wise manner, and call the extended logical language *Augmented Presburger Arithmetic* (APA). The decidability of Presburger Arithmetic, as well as Cooper's Algorithm, can be naturally extended to APA. Intuitively, verifying the equivalence of tuple expressions can be done by verifying the equivalence of their corresponding constituents.

Proposition 2. *The theory of formulas over $\mathbb{Z}^n$ with terms in the Augmented Presburger Arithmetic is decidable.*

Proof. Let φ be a quantified formula over $\bigcup_n \mathbb{Z}^n$ with terms in the Augmented Presburger Arithmetic. We shall translate φ to a formula in the Presburger Arithmetic. For any atom $A\colon = a = b$, where $a, b \in \mathbb{Z}^k$ for some $k > 0$, we build the following formula: $\bigwedge_{i=1}^{k} p_i(a) = p_i(b)$ and replace it in place of A. In the resulting formula, we assign new variable names, replacing the projected tuple variables: For $a \in \mathbb{Z}^k$ we define $x_{a,i} = p_i(a)$ for $i \in \{1, \ldots, k\}$. Variable quantification extends naturally, i.e. $\forall a$ becomes $\forall x_{a,1}, \ldots, x_{a,k}$, and similarly for $\exists$.

To be compatible with SparkLite's requirements, it will be useful to discuss an extension of APA in which terms are allowed to contain two additional constructs: ite expressions, and $\bot$ values. We denote this extension APA$^+$, and show how formulas in APA$^+$ can be converted to APA formulas.

The program terms may contain ite and $\bot$ expressions, therefore we need to encode the formulas in APA. We present a translation procedure $\mathcal{N}$ for converting

[11] We assume the reader is familiar with FOL, and omit a more formal description.

[12] Originally, Presburger Arithmetic was defined as a theory over natural numbers. However, its extension to integers and booleans is also decidable. (See, e.g., [2].)

[13] We denote infinities as $+\infty, -\infty \in \mathbb{Z}$.

[14] The complexity of Cooper's algorithm is $O(2^{2^{2^{pn}}})$ for some $p > 0$ and where n is the number of symbols in the formula [23].

Arithmetic Expression $ae ::= c \mid v \mid ae + ae \mid -\, ae \mid c * ae \mid ae \,/\, c \mid ae \,\%\, c$
Boolean Expression $be ::= \texttt{true} \mid \texttt{false} \mid b \mid e = e \mid ae < ae \mid \neg be \mid be \wedge be \mid be \vee be$
Primitive Expression $pe ::= ae \mid be$
Basic Expression $e ::= pe \mid \boldsymbol{v} \mid (pe \,, \overline{pe}) \mid \boldsymbol{p}_i(e) \mid \texttt{ite}(be, e, e)$

c, v, and b denote integer numerals, integer variables, and boolean variables, respectively. % denotes the modulo operator.

Fig. 6. Terms of the Augmented Presburger Arithmetic

APA$^+$ formulas to APA. Let φ be a formula. Following the standard notation of *sub-terms*, *positions*, and *substitutions* in [25],[15] we use $\varphi|_p$ to denote the *sub-term* of φ in a specific *position* p and by $\varphi[r]_p$ the substitution of the sub-term in position p with r. We use this notation to define $\mathcal{N}$. If $\varphi|_p = ite(\varphi_1, \varphi_2, \varphi_3)$, then φ is converted to: $(\varphi_1 \implies \varphi[\varphi_2]|_p) \wedge (\neg\varphi_1 \implies \varphi[\varphi_3]|_p)$. In addition, for every sub-term of the form $\varphi|_p = f(t_1, \ldots, t_n)$, if some t_i is equal (syntactically) to $\perp$, then $\varphi|_p = \perp$, as there is no meaning to evaluating functions on $\perp$ symbols, which represent non-existing RDD elements. Finally, we replace $\perp = \perp, x \neq \perp$ with tt, and $\perp \neq \perp, x < \perp, x \leq \perp, x > \perp, x \geq \perp, x = \perp$ with $f\!f$. We define a translation function $\mathcal{N}(\varphi)$, which goes over all positions in φ and performs substitutions as above. For example, $\varphi = (ite(x>0, x, \perp) = \perp)$ is translated to: $((x>0 \implies f\!f) \wedge (x \leq 0 \implies tt))$. Indeed, both $\varphi, \mathcal{N}(\varphi)$ are true only for $x \leq 0$.

Proposition 3 (φ and $\mathcal{N}(\varphi)$ are equivalent). *For every APA$^+$ formula φ, the APA formula $\varphi' = \mathcal{N}(\varphi)$, received by replacing all ite sub-terms with two implication conjuncts, all function calls with $\perp$ arguments to $\perp$, and all equalities and inequalities containing a $\perp$ symbol with either tt or $f\!f$, is equivalent to φ: $\varphi \iff \mathcal{N}(\varphi)$*

B Proof of Proposition 1

Proof. By symmetry, we assume without loss of generality $t_1 \neq \perp$. Therefore, there is an element in the RDD defined by t_1: $\exists \overline{x}, y. y = t_1(\overline{x}) \wedge y \neq \perp$. Denoting $\overline{x} = (x_1, \ldots, x_l)$, we choose input RDDs $\overline{r}$ such that each input RDD has a single element x_i of multiplicity n_i: $r_i = \{\!\{x_i; n_i\}\!\}$, for $i = 1, \ldots, l$. If $t_2(\overline{x}) \neq y$ then $[\![t_1]\!](\overline{r}) \neq [\![t_2]\!](\overline{r})$, as required. Otherwise, the multiplicity of y in $[\![t_1]\!]$ is $\mathit{\Pi}_{\mathbf{x}_{r_i} \in FV(t_1)} n_i$, and in $[\![t_2]\!]$ it is $\mathit{\Pi}_{\mathbf{x}_{r_i} \in FV(t_2)} n_i$. As $FV(t_1) \neq FV(t_2)$, there are $n_i > 1$ such that $\mathit{\Pi}_{\mathbf{x}_{r_i} \in FV(t_1)} n_i \neq \mathit{\Pi}_{\mathbf{x}_{r_i} \in FV(t_2)} n_i$, thus $[\![t_1]\!](\overline{r}) \neq [\![t_2]\!](\overline{r})$, as required.

C Proof of Theorem 1

Proof. For non-RDD return types, the absence of aggregate operators implies we can use Proposition 2, as the returned expression is expressible in APA. For RDD return type, we use the algorithm in Figure 5, which is a decision procedure:

[15] For brevity, we omit the technical details of these standard definitions.

- If both program terms evaluate to the empty bag for any choice of input RDDs, the algorithm detects it and outputs the programs are equivalent.
- Otherwise, the algorithm checks syntactically that $FV(\Phi(P_1)) = FV(\Phi(P_2))$. If that is not the case, then by Proposition 1 we can conclude the programs are not equivalent.
- The correctness of the algorithm in the next step follows from the semantics. If such an $\bar{x}$ is found, we take input RDDs $\bar{R}$, where $R_i = \{\!\{x_i; 1\}\!\}$, for which $[\![\Phi(P_i)]\!](\bar{R}) = \Phi(P_i)[\bar{x}/FV(\Phi(P_i))]$, thus $[\![\Phi(P_1)]\!](\bar{R}) \neq [\![\Phi(P_2)]\!](\bar{R})$. Otherwise, the formula is unsatisfiable. As $FV(\Phi(P_1)) = FV(\Phi(P_2))$ from the previous step, we know that the additional multiplicity donated by any valuation $\bar{x} \in \bar{R}$ is equal to $\Pi_i R_i(x_i)$ in both programs. We conclude that for all possible choices of input RDDs, the resulting bags have the same elements, and those elements have the same multiplicities in each bag, as required.

D Proof of Theorem 2

Proof. We show a reduction of Hilbert's 10^{th} problem to *PE*. Hilbert's 10^{th} problem is the problem of finding a general algorithm that given a polynomial with integer coefficients, decides whether it has integer roots. We assume towards a contradiction that *PE* is decidable. Let there be a polynomial p over k variables $x_1, \ldots, x_k$, and coefficients $a_1, \ldots, a_n$. We use SparkLite operations and input RDDs R_i to represent the value of the polynomial p for some valuation of the x_i. We define a translation φ from monomials to SparkLite expressions [16]. Note, that we allow to nest RDD operations inside other RDD operations, which while not being explicitly allowed according to the SparkLite syntax, can be readily formulated properly as a series of 'let' expressions.

- $\varphi(x_i) = R_i$
- $\varphi(x_{i_1}^{n_1} \cdots x_{i_l}^{n_l}) = \texttt{cartesian}(R_{i_1}, \varphi(x_{i_1}^{n_1-1} \cdots x_{i_l}^{n_l}))$ ($n_j > 0$ for $j = 1, \ldots, l$)

In addition, given a monomial m, we define $\hat{\varphi}(m) = \texttt{fold}(0, \lambda A, \bar{x}.A + 1)(\varphi(m))$. Given a polynomial $p(a_1 \ldots, a_n; x_1, \ldots, x_k) = \sum_{l=1}^{n} a_l m_l$ where m_l are monomials over $x_1, \ldots, x_k$, we generate the following instance of the *PE* problem:

```
P1($R_1, \ldots, R_k$: $RDD_{\texttt{Int}}$):        P2($R_1, \ldots, R_k$: $RDD_{\texttt{Int}}$):
return $\sum_{l=1}^{n} a_l \hat{\varphi}(m_l) \neq 0$          return $tt$
```

By choosing input RDDs such that the size of R_i is equal to the matching variable x_i, we can simulate any valuation to the polynomial p. If $P1$ returns

[16] Note we allow self cartesian products, i.e. expressions like $\texttt{cartesian}(R, R)$. It is possible to have an equivalent reduction which is not using self cartesian products, by representing each variable x_i, whose highest power in the polynomial is p_i, using p_i RDDs. The output program will first verify that for any variable x_i, all p_i RDDs representing x_i have the same size (this can be done using the *fold* operation and comparison of the results). If not, the program returns true. Otherwise, any power of x_i up to p_i will be represented using a cartesian product of a subset of the different RDDs of x_i. The rest of the reduction's details are the same as in this reduction.

true, then the valuation is not a root of the polynomial p. Thus, if it is equivalent to the 'true program' $P2$, then the polynomial p has no roots. Therefore, if the algorithm solving PE outputs 'equivalent' then the polynomial p has no roots, and if it outputs 'not equivalent' then the polynomial p has some root, where $x_i = |[\![R_i]\!]|$ for the R_i's which serve as the witness for nonequivalence. Thus we have a reduction of Hilbert's 10^{th} problem, proving PE is undecidable.

E Proof of Lemma 1

Proof. (Lemma 1) First we recall the semantics of the `fold` operation on some RDD R, which is a bag. We choose an arbitrary element $a \in R$ and apply the fold function recursively on a and on R with a single instance of a removed. We then write a sequence of elements in the order they are chosen by `fold`: $\langle a_1, \ldots, a_n \rangle$, where n is size of the bag R. We also know that a requirement of aggregating operations' UDFs is that they are *commutative*, so the order of elements chosen does not change the final result. We also recall we extended f_i to $\xi_i \times (\sigma_i \cup \{\bot\})$ by setting $f_i(A, \bot) = A$ ($\bot$ is defined to behave as the neutral element for f_i). We denote $[\![\varphi_1]\!] = R_1, [\![\varphi_2]\!] = R_2$. To prove $g_1([\varphi_1]_{init_1,f_1}) = g_2([\varphi_2]_{init_2,f_2})$, it is necessary to prove that

$$g_1([\![\texttt{fold}]\!](f_1, init_1)(R_1)) = g_2([\![\texttt{fold}]\!](f_2, init_2)(R_2))$$

We set $A_{\varphi_j,0} = init_j$ for $j = 1, 2$. Each element of R_1 and R_2 is expressible by providing a concrete valuation to the free variables of φ_1, φ_2, namely the vector $\bar{v}$. We prove the equality by induction on the *size* of the RDDs R_1, R_2, denoted n.[17] We choose an arbitrary sequence of n valuations $\langle \bar{a}_1, \ldots, \bar{a}_n \rangle$, and plug them into the *fold* operation for both R_1, R_2. The result is two sequences of *intermediate values* $\langle A_{\varphi_1,1}, \ldots, A_{\varphi_1,n} \rangle$ and $\langle A_{\varphi_2,1}, \ldots, A_{\varphi_2,n} \rangle$. From the semantics of `fold`, we have that $A_{\varphi_j,i} = f_j(A_{\varphi_j,i-1}, \varphi_j(\bar{a}_i))$ for $j = 1, 2$. Our goal is to show $g(A_{\varphi_1,n}) = g'(A_{\varphi_2,n})$ for all n.

Case $n = 0$: $R_1 = R_2 = \{\!\{\}\!\}$, so $[\![\texttt{fold}]\!](f_1, init_1)(R_1) = init_1$ and $[\![\texttt{fold}]\!](f_2, init_2)(R_2) = init_2$. From Equation (2), $g_1(init_1) = g_2(init_2)$, as required.

Case $n = i$, assuming correct for $n \leq i - 1$: By assumption, we know that the sequence of intermediate values up to $i-1$ satisfies: $g_1(A_{\varphi_1,i-1}) = g_2(A_{\varphi_2,i-1})$. We are given the i'th valuation, denoted $\bar{a}_i$. We need to show $A_{\varphi_1,i} = A_{\varphi_2,i}$, so we use the formula for calculating the next intermediate value:

$$\begin{aligned} A_{\varphi_1,i} &= f_1(A_{\varphi_1,i-1}, \varphi_1(\bar{a}_i)) \\ A_{\varphi_2,i} &= f_2(A_{\varphi_2,i-1}, \varphi_2(\bar{a}_i)) \end{aligned}$$

We use Equation (3), plugging in $\bar{v} = \bar{a}_i$, $A_{\varphi_1} = A_{\varphi_1,i-1}$, and $A_{\varphi_2} = A_{\varphi_2,i-1}$. By the induction assumption, $g_1(A_{\varphi_1,i-1}) = g_2(A_{\varphi_2,i-1})$, therefore $g_1(A_{\varphi_1}) =$

[17] It is important to note that not every n can be a legal size of the RDDs. For example, if $R_1 = \texttt{cartesian}(R, R)$, then its size must be quadratic ($|R|^2$). The induction we apply here, is actually stronger than what is required for equivalence, because we prove the equivalence even for subsets of the RDDs which may not be expressible using SparkLite operations. In any case, the soundness argument is valid.

$g_2(A_{\varphi_2})$, so Equation (3) yields $g_1(f_1(A_{\varphi_1}, \varphi_1(\bar{a}_i))) = g_2(f_2(A_{\varphi_2}, \varphi_2(\bar{a}_i)))$. By substituting back A_{φ_j} and the formula for the next intermediate value, we get: $g_1(A_{\varphi_1,i}) = g_2(A_{\varphi_2,i})$ as required.

F Proof Theorem 3

Proof. **Sound (if):** We prove the equality $g_1([\varphi_1]_{init_1,f_1}) = g_2([\varphi_2]_{init_2,f_2})$ by induction on the size of the RDDs $[\![\varphi_1]\!], [\![\varphi_2]\!]$, denoted n.[18] For $n = 0$, $[\![\varphi_1]\!](\bar{r}) = [\![\varphi_2]\!](\bar{r}) = \{\!\!\{\}\!\!\}$, thus $[\varphi_i]_{init_i,f_i} = init_i$ $(i = 1, 2)$, and the equality follows from Equation (6). Assuming for n and proving for $n{+}1$: We let a sequence of intermediate values $A_{\varphi_i,k}$, $(i = 1, 2; k = 1, \ldots, n{+}1)$, for which we know in particular that $g_1(A_{\varphi_1,n}) = g_2(A_{\varphi_2,n})$, and we need to prove $g_1(A_{\varphi_1,n+1}) = g_2(A_{\varphi_2,n+1})$. We denote $A_{\varphi_i,0} = init_i$, and then we have $A_{\varphi_i,k} = f_i(A_{\varphi_i,k-1}, \varphi_i(\bar{a_k}))$ $(k = 1, \ldots, n + 1)$ for some $\bar{a_k}$. According to Equation (5), $A_{\varphi_i,2} = f_i(A_{\varphi_i,1}, \varphi_i(\bar{a_2})) = f_i(f_i(init_i, \varphi_i(\bar{a_1})), \varphi_i(\bar{a_2}))$ yields $\exists \bar{a_2}'. \bigwedge_{i=1,2} A_{\varphi_i,2} = f_i(init_i, \varphi_i(\bar{a_2}'))$. We can thus use Equation (5) to prove by induction that $\exists \bar{a_k}'. \bigwedge_{i=1,2} A_{\varphi_i,k} = f_i(init_i, \varphi_i(\bar{a_k}'))$, and in particular $\exists \bar{a_n}'. \bigwedge_{i=1,2} A_{\varphi_i,n} = f_i(init_i, \varphi_i(\bar{a_n}'))$. By applying Equation (7) for $\bar{v} = \bar{a_{n+1}}, \bar{y} = \bar{a_n}'$, we get:

$$\begin{aligned}
g_1(f_1(f_1(init_1, \varphi_1(\bar{y})), \varphi_1(\bar{v}))) &= g_2(f_2(f_2(init_2, \varphi_2(\bar{y})), \varphi_2(\bar{v}))) &&\Longrightarrow\\
g_1(f_1(f_1(init_1, \varphi_1(\bar{a_n}')), \varphi_1(\bar{a_{n+1}}))) &= g_2(f_2(f_2(init_2, \varphi_2(\bar{a_n}')), \varphi_2(\bar{a_{n+1}}))) &&\Longrightarrow\\
g_1(f_1(A_{\varphi_1,n}, \varphi_1(\bar{a_{n+1}}))) &= g_2(f_2(A_{\varphi_2,n}, \varphi_2(\bar{a_{n+1}}))) &&\Longrightarrow\\
g_1(A_{\varphi_1,n+1}) &= g_2(A_{\varphi_2,n+1})
\end{aligned}$$

as required.

Complete (only if): Assume towards a contradiction that either Equation (6) or Equation (7) are false. If the requirement of Equation (6) is not satisfied, yet the aggregates are equivalent, i.e.

$$g_1([\varphi_1]_{init_1,f_1}) = g_2([\varphi_2]_{init_2,f_2}) \wedge g_1(init_1) \neq g_2(init_2)$$

then we can get a contradiction by choosing all input RDDs to be empty. Thus, for $R = \{\!\!\{\}\!\!\}$, $[\![[\varphi_1]_{init_1,f_1}]\!](R) = init_1 \wedge [\![[\varphi_2]_{init_2,f_2}]\!](R) = init_2 \implies g_1(init_1) = g_2(init_2)$, which is a contradiction. The conclusion is that Equation (6) is a necessary condition for equivalence. Therefore, we assume just Equation (7) is false. Let there be counter-examples $\bar{v}, \bar{y}$ to Equation (7), [19] and let:

$$F_i = f_i(f_i(init_i, \varphi_i(\bar{y})), \varphi_i(\bar{v}))$$

Then $g_1(F_1) \neq g_2(F_2)$. By Equation (5) we can write F_i as: $F_i = f_i(init_i, \varphi_i(\bar{w}))$ for some $\bar{w}$. We take an RDD $R = \{\!\!\{\bar{w}; 1\}\!\!\}$. Then $[\![\varphi_j]\!](R) = \{\!\!\{\varphi_j(\bar{w}); 1\}\!\!\}$, for which: $[\![[\varphi_j]_{init_j,f_j}]\!](R) = F_i$. By the assumption, $[\![g_1([\varphi_1]_{init_1,f_1})]\!](R) = [\![g_2([\varphi_2]_{init_2,f_2})]\!](R)$, but then $g_1(F_1) = g_2(F_2)$. Contradiction.

[18] The comment in footnote 17 regarding the validity of the soundness argument, even if $[\![\varphi_i]\!]$ can not have size n, is still valid here.

[19] Note that the A_{φ_i} are determined immediately by choosing $\bar{y}$: $A_{\varphi_i} = f_i(init_i, \varphi_i(\bar{y}))$.

G Proof of Lemma 2

Proof. The proof follows along the lines of the proof of Lemma 1. We need to prove $\Phi(P_1) = \Phi(P_2)$, or $\forall \bar{x}.\psi_1[\gamma_1]|_{p_1}(\bar{x}) = \psi_2[\gamma_2]|_{p_2}(\bar{x})$, where $\gamma_i = [\varphi_i]_{init_i,f_i}$ and $\bar{x}$ is a vector of valuations to $FV(\psi_1), FV(\psi_2)$ which are equal sets (Equation (9)). We shall prove it by induction on the size of the RDDs R_1, R_2, generating the underlying terms of γ_1, γ_2.

For size 0, we have $\gamma_i = init_i$, and from Equation (10) we have $\Phi(P_1) = \Phi(P_2)$ as required.

Assuming for size n and proving for $n + 1$: The RDDs R_1, R_2 are now generated using $a_1, \ldots, a_{n+1}$, with intermediate values $A_{\varphi_i,1}, \ldots, A_{\varphi_i,n+1}$ for $i = 1, 2$. By assumption, $\forall x.\psi_1[A_{\varphi_1,n}]|_{p_1} = \psi_2[A_{\varphi_2,n}]|_{p_2}$, and we need to prove $\forall \bar{x}.\psi_1[A_{\varphi_1,n+1}]|_{p_1}(\bar{x}) = \psi_2[A_{\varphi_2,n+1}]|_{p_2}(\bar{x})$. In addition, $A_{\varphi_i,n+1} = f_i(A_{\varphi_i,n}, a_{n+1})$ for $i = 1, 2$. We let some $\bar{x}$ and we need to prove for it $\psi_1[A_{\varphi_1,n+1}]|_{p_1}(\bar{x}) = \psi_2[A_{\varphi_2,n+1}]|_{p_2}(\bar{x})$. We apply Equation (11) with $\bar{x}$ as $\bar{x}$, $\bar{v} = a_{n+1}$, and $A_{\varphi_1,n}, A_{\varphi_2,n}$ as A_1, A_2, concluding that: $\psi_1[f_1(A_{\varphi_1,n}, a_{n+1})]|_{p_1}(\bar{x}) = \psi_2[f_2(A_{\varphi_2,n}, a_{n+1})]|_{p_2}(\bar{x})$. Replacing for $A_{\varphi_i,n+1}$, we get what had to be proven.