# Representing MapReduce Optimisations in the Nested Relational Calculus

Marek Grabowski[1], Jan Hidders[2], and Jacek Sroka[1]

[1] Institute of Informatics, University of Warsaw, Poland
{gmarek, sroka}@mimuw.edu.pl
[2] Delft University of Technology, The Netherlands
a.j.h.hidders@tudelft.nl

**Abstract.** The MapReduce programming model is recently getting a lot of attention from both academic and business researchers. Systems based on this model hide communication and synchronization issues from the user and allow processing of high volumes of data on thousands of commodity computers. In this paper we are interested in applying MR to processing hierarchical data with nested collections such as stored in JSON or XML formats but with restricted nesting depth as is usual in the nested relational model. The current data analytics systems now often propose ad-hoc formalisms to represent query evaluation plans and to optimize their execution. In this paper we will argue that the Nested Relation Calculus provides a general, elegant and effective way to describe and investigate these optimizations. It allows to describe and combine both classical optimizations and MapReduce-specific optimizations. We demonstrate this by showing that MapReduce programs can be expressed and represented straightforwardly in NRC by adding syntactic short-hands. In addition we show that optimizations in existing systems can be readily represented in this extended formalism.

## 1 Introduction

MapReduce (MR) is a programming model developed at Google to easily distribute processing of high volumes of data on thousands of commodity computers. Systems based on this model hide communication and synchronization issues from the user, while enforcing a simple yet powerful programming style which is influenced by functional programming. MR is being successfully applied [6] on Web scale data at Google processing centers. After Google published the paper explaining the idea behind MR, an open source version, named Hadoop [2], was created and started to be widely used by both universities for research and companies like Yahoo, Twitter and Facebook to process their data.

Even though the MR model makes writing distributed, data-driven software a lot easier than with older technologies like MPI or OpenMP, for many applications it is too general and low level. This forces the developers who want to process large collections of data to deal with multiple concerns. Additionally to dealing with the problem they are trying to solve, they have to struggle with implementing and optimizing typical operations. As happened many times in history, e.g., with compilers or relational databases, it is better to separate concerns by introducing a higher level, more declarative language, better suited for specifying tasks for data analytics or scientific data processing, and at the same time more amenable to optimization. This is a topic of intensive study at the time of this writing and many systems are being built on top of implementations of MR ranging from data analytics Pig [10], data warehousing Hive [16], through workflow systems like Google's Sawzall [15], to graph processing systems like Pregel [14].

An effort is also undertaken on finding the best formal model of MR computations and their cost that would allow to reason which algorithms can be efficiently expressed in MR, and which cannot and why. Several attempts were made to define cost model, which is easy to use, understand and allows to reason on MR programs efficiency. One of the more successful ideas is a notion of *replication rate* introduced by Afrati, Ullman et al. [1], who count the number of excessive data transfers and claim that it is a good metric of MR efficiency since it deals with the bottleneck of typical MR programs — the network efficiency. In another work by Karloff et al. [12] a notion of MR expressive power was researched and a correspondence showed between MR framework and a subclass of PRAM.

In this paper we are interested in applying MR to processing hierarchical data with nested collections such as stored in JSON or XML formats but with restricted nesting depth as is usual in the nested relational model. We show that the Nested Relational Calculus (NRC) [3], a core language that describes a small but practically and theoretically interesting class of queries and transformations over nested relational data, is a good formalism to reason about MR programs. We demonstrate this by showing how some of the most important higher-level MR languages can be expressed in NRC with added syntactic short-hands. In addition we show that NRC provides a general, elegant and effective way to describe and investigate optimizations used in existing systems.

A similar attempt was made by Lämmel [13], who expressed the MR framework in t he Haskell language. Our approach is more formal and slightly stronger, as we are using a more minimal and formally defined calculus which is not Turing complete.

## 2 The Nested Relational Calculus: NRC

In this paper we are interested in operations on large datasets, mainly representing large collections of data. For simplicity we will restrict ourselves to one collection type: bags. This collection type has more algebraic properties than lists, which can be exploited for optimization purposes, and the natural notion of grouping is easily represented in terms of bags. Moreover, sets can be introduced straightforwardly by means of a **set** operator which removes duplicates from a bag. We denote bags by generalizing set enumeration, and so $\{a, a, b\}$ denotes the bag that contains $a$ twice and $b$ once. The bag union is denoted as $\uplus$

and is assumed to be additive, e.g., $\{a, b\} \uplus \{b, c, c\} = \{a, b, b, c, c\}$. In this paper we consistently use curly brackets $\{\ \}$ to denote bags, not sets as usual.

Our data model will be essentially that of the nested relational model. Allowed data types are (1) *basic types* which contain *atomic values* being constants from some domain, which is assumed here to include at least integers and booleans, and to allow equality tests (2) named tuples with field names being strings and values being of any of allowed data types and (3) bag types which describe finite bags containing elements of a specify type. The instances of types will be called both *data* and *values* interchangeably.

For the purpose of optimization we will focus on a limited set of operators and language constructs that is a variant of the NRC. We will use the following syntax for our variant of NRC:

$$\begin{aligned} E ::= {} & C \mid X \mid \emptyset \mid \{E\} \mid E \uplus E \mid \langle K : E, \ldots, K : E\rangle \mid E.K \mid \\ & \mathbf{set}(E) \mid F(E) \mid \{E \mid X \in E, \ldots, X \in E, E, \ldots, E\} \mid E \approx E. \end{aligned}$$

We discuss the constructs in the order of appearance. The first one is $C$ which describes denotations of constant atomic values. The nonterminal $X$ stands for variables. The following three constructs form the basic bag operations, i.e., the empty bag constructor, the singleton bag constructor and the additive bag union. Usually also a typing regime is introduced with the calculus to ensure well-definedness and for example require that the bag union is only applied to bags of the same type, but for brevity we omit this and refer the reader to [17, 7].

Next we have the tuple creation and tuple projection. In this paper we will be working with named tuples. The nonterminal $K$ stands for field names and it must hold that all field names are distinct. The types that describe tuples, called *tuple types*, are defined as partial functions form field names to types, i.e., $\langle id : 1\rangle$ and $\langle value : 1\rangle$ have different types, but $\langle 1 : int, 2 : bool\rangle$ and $\langle 2 : bool, 1 : int\rangle$ have the same type. We will sometimes omit the column names. Notice that we allow the empty tuple as well, which will be called *unit* and denoted $\langle\rangle$.

The next construct is the **set** operator, which removes all duplicates from a bag. The $F$ represents *user defined functions* (UDFs). We require that UDFs come with a well defined semantics, being a partial function from all possible values to all possible values, and are describe by a type. The Nested Relational Calculus as presented here is parametrized with the set of user-defined functions $F$, which is denoted as NRC$^F$.

The bag comprehensions of the form $\{e \mid x_1 \in e_1, \ldots, x_n \in e_n, e'_1, \ldots, e'_m\}$, also known from some functional programming languages [18] and query evaluation languages [9], returns a bag, which is constructed in the following way: starting with $x_1 \in e_1$ and going to the right we iterate over the given collection (here $e_1$) and assign one element at the time to the given variable (here $x_1$) in a nested-loop fashion. In the body of the innermost loop the expressions $e'_1, \ldots, e'_m$ are evaluated, and if any of their values is **false** then no output is generated, otherwise $e$ is evaluated and its value added to the returned collection. Finally the construct of the form $e_1 \approx e_2$ denotes the equality between values and returns **true** if $e_1$ and $e_2$ evaluate to the same values, and **false** otherwise. The equality between tuples or bags is defined in a standard way.

Below we define the semantics for the NRC, which starts with the definition of substitution of variable $x$ with expression $e'$ in expression $e$, denoted as $e_{[x/e']}$. Its formal semantics can be defined by induction on the structure of $e$ as follows:

$$\frac{}{c_{[x/e]} = c} \qquad \frac{}{x_{[x/e]} = e} \qquad \frac{x \neq y}{x_{[y/e]} = x} \qquad \frac{}{\emptyset_{[x/e]} = \emptyset} \qquad \frac{}{\{e\}_{[x/e']} = \{e_{[x/e']}\}}$$

$$\frac{}{(e_1 \uplus e_2)_{[x/e']} = e_{1[x/e']} \uplus e_{2[x/e']}}$$

$$\frac{}{\langle \kappa_1 : e_1, \ldots, \kappa_n : e_n \rangle_{[x/e]} = \langle \kappa_1 : e_{1[x/e]}, \ldots, \kappa_n : e_{n[x/e]} \rangle} \qquad \frac{}{e.\kappa_{[x/e']} = e_{[x/e']}.\kappa}$$

$$\frac{}{\mathbf{set}(e)_{[x/e']} = \mathbf{set}(e_{[x/e']})} \qquad \frac{}{f(e)_{[x/e']} = f(e_{[x/e']})} \qquad \frac{}{\{e\ |\}_{[x/e']} = \{e_{[x/e']}\ |\}}$$

$$\frac{}{\{e \mid x_1 \in e_1, \Delta\}_{[x_1/e']} = \{e \mid x_1 \in e_{1[x_1/e']}, \Delta\}}$$

$$\frac{x_1 \neq y \qquad \{e \mid \Delta\}_{[y/e']} = \{e'' \mid \Delta''\}}{\{e \mid x_1 \in e_1, \Delta\}_{[y/e']} = \{e'' \mid x_1 \in e_{1[x_1/e']}, \Delta''\}}$$

$$\frac{\{e \mid \Delta\}_{[y/e']} = \{e'' \mid \Delta''\}}{\{e \mid e_1, \Delta\}_{[y/e']} = \{e'' \mid e_{1[x_1/e']}, \Delta''\}} \qquad \frac{}{(e_1 \approx e_2)_{[x/e']} = (e_{1[x/e']} \approx e_{2[x/e']})}$$

Now we define the NRC semantics, i.e., the relation $e \Rightarrow v$ which denotes that expression $e$ returns value $v$. It is defined in the following way:

$$\frac{}{c \Rightarrow c} \qquad \frac{}{\emptyset \Rightarrow \emptyset} \qquad \frac{e \Rightarrow v}{\{e\} \Rightarrow \{v\}} \qquad \frac{e \Rightarrow \{v_1, \ldots, v_n\}, e' \Rightarrow \{v'_1, \ldots, v'_n\}}{e \uplus e' \Rightarrow \{v_1, \ldots, v_n, v'_1, \ldots, v'_n\}}$$

$$\frac{e_1 \Rightarrow v_1, \ldots, e_n \Rightarrow v_n}{\langle \kappa_1 : e_1, \ldots, \kappa_n : e_n \rangle \Rightarrow \langle \kappa_1 : v_1, \ldots, \kappa_n : v_n \rangle} \qquad \frac{e \Rightarrow \langle \kappa_1 : v_1, \ldots, \kappa_n : v_n \rangle}{e.\kappa_i \Rightarrow v_i}$$

$$\frac{e \Rightarrow \{v_1, \ldots, v_n\}}{\mathbf{set}(e) \Rightarrow \cup_{i=1}^{n} \{v_i\}} \qquad \frac{f(v) \Rightarrow v' \qquad e \Rightarrow v}{f(e) \Rightarrow v'} \qquad \frac{e_1 \Rightarrow \mathbf{false}}{\{e \mid e_1, \Delta\} \Rightarrow \emptyset}$$

$$\frac{e_1 \Rightarrow \{v_1, \ldots, v_n\} \qquad \forall_{i=1}^{n} (\{e \mid \Delta\}_{[x_i/v_i]} \Rightarrow v'_i)}{\{e \mid x_1 \in e_1, \Delta\} \Rightarrow \uplus_{i=1}^{n} v'_i} \qquad \frac{e_1 \Rightarrow \mathbf{true} \qquad \{e \mid \Delta\} \Rightarrow v}{\{e \mid e_1, \Delta\} \Rightarrow v}$$

$$\frac{e \Rightarrow v}{\{e\ |\} \Rightarrow \{v\}} \qquad \frac{e \Rightarrow v \qquad e' \Rightarrow v' \qquad v \neq v'}{e \approx e' \Rightarrow \mathbf{false}} \qquad \frac{e \Rightarrow v \qquad e' \Rightarrow v}{e \approx e' \Rightarrow \mathbf{true}}$$

where we let $\oplus_{i=1}^{n} S_i$ denote $S_1 \oplus \ldots \oplus S_n$. Observe that the result of an expression is defined iff the expression contains no free variables.[3]

## 3 MapReduce

MapReduce (MR) is a programming model for heavily distributed software, which hides most of the complexity coming from parallelism. The system handles communication, synchronization and failure recovery, while the user is responsible only for writing the program logic in the form of Map and Reduce functions. The working of the system is described in detail in [6], below we give only an outline of its design.

The input of a MR routine is a collection of key-value pairs. The main assumption is that the collection is too big to fit into the memory of a machine, so it is necessary to distribute the computations over multiple machines. MR is built on top of a Distributed File System (DFS) (for example the Google File System [11]) and takes advantage of its architecture, as the input is stored in DFS, thus it is divided into blocks, spread and replicated throughout a cluster.

The first stage of an MR routine is the *Map* phase. In the ideal case there is almost no communication needed during this phase, as the system tries to process the data on machines that already store it, which is often feasible thanks to the replication. In real life some of the data may need to be sent, but we chose to ignore it, as it is too low level for our model and the amount of necessary communication depends on many hard to predict factors, like cluster configuration and its load. After the map phase, the user can choose to run the combine phase, which takes all the data from a single mapper for a single key and runs a UDF on such a collection. As this phase is designed only to improve efficiency not the expressive power of the model, e.g. some frameworks may choose not execute *Combine* functions, we chose to skip this phase all together.

The next stage of MR is opaque for the user and is called the *Shuffle* phase. It consists of grouping all the *Map* outputs which have the same intermediate key and sending the data to the machines on which *Reduce* functions will be run. In practice there is a sort order imposed on the intermediate keys, and sometimes also on the grouped data, but we choose to ignore the order and work on bags instead, since the order is rarely relevant at the conceptual level of the transformation, i.e., users usually think about their collections as bags and do not care about the specific ordering. This is the stage where communication and synchronization takes place, and opaqueness of it makes the reasoning about MR routines easier. In some implementations of MR the user can specify a partitioner, which is responsible for distributing the data between machines running the reducers. Note that this behavior may be modeled using secondary-key grouping, as it is required that all datagrams with the same intermediate key end up on the same machine.

[3] Note that we do not require that the free variables of the substituted expression are not bound after the substitution, since we only substitute values.

The last phase is called the *Reduce* phase, and it consists of independent executions of the *Reduce* function, each on a group of values with the same intermediate-key, and produces a collection of *result* key-value pairs.

It is possible to feed a Reducer from multiple different mappers, as long as the *Shuffle* phase can group the outputs of all the mappers together. In such a case, the intermediate data from all mappers is treated identically and is merged together for shuffling. Furthermore, often MR routines are pipelined together, making the output of one routine, an input of another one. In such a case product-keys of the former, become input-keys of the latter. Sometimes by MR we do not mean a MR construct, but a computation model consisting of MR routines and ability to connect those routines to form a DAG. Such computation model is parametrized with a class of allowed Map and Reduce UDFs. It should be clear from the context which meaning of MR we have in mind.

We chose a simplified version of MR, without ordering, intermediate *Combine* phase, Partitioning etc., as this is the model appearing most often in the literature. Furthermore some of our simplifications do not impact the expressive power of the model, which what we are interested in this paper. Those simplifications may turn out to be too strong in the future, to work with some low-level query optimizations, but they proved to be appropriate for the optimizations we are considering in this paper.

## 4 Defining MapReduce in NRC

We proceed by showing that the MR framework can be defined using NRC constructs described in the previous section. Here we want the reader to note that both **Map** and **Reduce** phases apply *Map* and *Reduce* UDFs to the data. In the general case functions passed as arguments to **Map** and **Reduce** can be arbitrary, as long as they have following types: $\langle k : \alpha_1, v : \beta_1\rangle \to \{\langle k : \alpha_2, v : \beta_2\rangle\}$ and $\langle k : \alpha_2, vs : \{\beta_2\}\rangle \to \{\langle k : \alpha_3, v : \beta_3\rangle\}$, respectively. Here in the rest of the paper we use the short names $k$, $v$ and $vs$ for *key*, *value* and *values* respectively, to make the presentation shorter.

Note that in our definition of NRC there are no functions per se, so in stead we use expressions with free variables. To denote expressions with abstracted variables we use the $\lambda$ notation, e.g. if $e$ is a NRC expression with a single free variable $x$, with the semantics well defined for $x$ of a type $\alpha$ with a result type $\beta$, then $\lambda x.e$ is a function of type $\alpha \to \beta$.

The **Map** routine, where the first argument is a *Map* UDF, and $D$ is a phase input of type $\{\langle k : \alpha_1, v : \beta_1\rangle\}$, can be written in NRC as:

$$\textbf{Map}\quad [\lambda x.e_{map}](D) = \{z \mid y \in D, z \in e_{map[x/y]}\}.$$

Note that we assume that the result of $\lambda x.e_{map}$ is a collection and the **Map** flattens the result, hence the output of the **Map** has the same type as $\lambda x.e_{map}$.

The *Shuffle* phase at the conceptual level essentially performs a grouping on the key values. It can be expressed in NRC as:

$$\mathbf{Shuffle}(D) = \{\langle k : x, vs : \{z.v \mid z \in D, z.k \approx x\}\rangle \mid x \in \mathbf{set}(\{y.k \mid y \in D\})\},$$

The result of the *Shuffle* phase is a collection of key-collection pairs of all values grouped by keys.

The *Reduce* phase gets the output of the *Shuffle* phase, which is of type collection of key-collection pairs, and is responsible for producing the result of the whole MapReduce routine. It can be formulated in NRC as:

$$\mathbf{Reduce}\quad[\lambda x.e_{red}](D) = \{z \mid y \in D, z \in e_{red[x/y]}\},$$

Having defined all phases of MR separately, we can define the whole **MR** syntactic short-hand:

$$\mathbf{MR}\quad[\lambda x.e_{map}][\lambda x.e_{red}](D) = \mathbf{Reduce}\ [\lambda x.e_{red}](\mathbf{Shuffle}\,(\mathbf{Map}\ [\lambda x.e_{map}](D))),$$

The extension of NRC with the **MR** syntactic short-hand construct will be referred to as **NRC-MR**. Our definition or MR construct allows only one mapper. It is easy to generalize it to handle multiple mappers, as long as they have a common output type. To do so we need to feed the **Shuffle** with the union of all mapper outputs.

We define a **MR** program informally as a workflow described by a directed acyclic graph that ends with a single node and where each internal and final node with $n$ incoming edges as associated with (1) an **MR** step with $n$ mappers and (2) an ordering on the incoming edge. Moreover, the start nodes with no incoming edges are each associated with a unique input variable. When executing this program the data received through the $i$th input edge is fed to the $i$th mapper of that node. Note that our definition of **MR** program is valid for standard data processing, where **MR** is a top-level language. Sometimes, e.g. in case of workflows with feedback or graph algorithms, there is an additional level of programming needed on top of **MR** which introduces recursion. Our model can be seen as a formalization of **MR** programs where the dataflow does not depend on the actual data, which is the case for most of database queries.

**Theorem 1.** *Any MR program where the* **MR** *steps use as mappers and reducers functions expressible in NRC$^F$ can be expressed in NRC$^F$.*

*Proof.* Indeed any MR routine using functions from $F$ can be written in NRC using the **MR** shorthand, as showed above. Composing the routines into a DAG is equivalent to nesting the expressions for corresponding routines, in NRC.

## 5 Higher-level MapReduce-based languages

Recently, high-level languages compiled to MR are receiving a lot of attention. Examples of those attempts are Facebook's Hive [16] – a Hadoop based data warehouse with SQL-like query language, Yahoo!'s Pig Latin [10] – a data analysis imperative language with SQL-like data operations, and Fegaras's MRQL [8] – an extension of SQL, which allows rnesting and in which MR is expressible in the same sense as in the NRC. In this section we review the ideas and operators from those languages and also provide an overview of the types of optimizations their implementations include.

## 5.1 Hive QL

Hive [16], is designed to replace relational databases, so some of its features like data insertion are orthogonal to our perspective. The Hive compilation and execution engine uses knowledge of the physical structure of the underlying data store. Since we abstract from a concrete physical representation of the data, we concentrate on the Hive Query Language (Hive QL). Its SQL-based syntax allows subqueries in the *FROM* clause, equi-joins, group-by's and including MR code. Hives does a handful of optimizations which are applied while creating the execution plan, including: (1) combining multiple *JOIN*s on the same key into a single multi-way join,(2) pruning unnecessary columns from the data, (3) performing map-side *JOIN*s when possible and (4) tricks based on knowledge of the physical structure of the underlying data store.

## 5.2 Pig Latin

Pig Latin [10] is a query language for the Pig system. It is business intelligence language for parallel processing huge data sets. The data model of Pig is similar to the one in this paper, with nesting and data types like tuples, bags and maps. Unlike other languages discussed here, Pig Latin is not declarative. Programs are series of assignments, and similar to an execution plan of a relational database. The predefined operators are iteration with projection *FOREACH-GENERATE*, filtering *FILTER*, and *COGROUP*. The *COGROUP*s semantics is similar to a *JOIN* but instead of flattening the product, it leaves the collections nested, e.g., **COGROUP** *People* **BY** *address*, *Houses* **BY** *address* returns a collection of the type: $\langle address, \{People\ with\ given\ address\}, \{Houses\ with\ given\ address\}\rangle$. It is easy to see that *GROUP* is a special case of *COGROUP* where the input is a single bag, and *JOIN* is a *COGROUP* with a flattened result. In addition Pig Latin also provides the user with some predefined aggregators, like *COUNT*, *SUM*, *MIN* etc., which we skip in our work since their optimized implementation is a research topic on its own and requires the inclusion of the *Combine* phase.

On the implementation and optimization side, the Pig system starts with an empty logical plan and extends it one by one with the user-defined bags, optimizing the plan after each step. Pig generates a separate MR job for every *COGROUP* command in the expression. All the other operations are split between those *COGROUP* steps and are computed as soon as possible, i.e. operations before the first *COGROUP* are done in the very first Map step, and all the others in the reducer for the preceding *COGROUP*.

## 5.3 MRQL

MRQL is a query language designed by Fegaras et al. [8] as a declarative language for querying nested data, which is as expressive as MR. The language is also designed to be algebraic in the sense that all the expressions can be combined in arbitrary ways. MRQL expressions are of the form:

$$\textbf{select } e \textbf{ from } d_1 \textbf{ in } e_1, d_2 \textbf{ in } e_2, \ldots, d_n \textbf{ in } e_n$$
$$[\textbf{where } pred]\ [\textbf{group\_by } p' : e'\ [\textbf{having } e_h]]\ [\textbf{order\_by } e_o\ [\textbf{limit } n]]$$

where $e$'s denote nested MRQL expressions.

What is the most interesting in *MRQL* from our perspective, is not the language itself since it is similar to SQL, but the associated underlying physical algebraic operators. The main two operators are *groupBy* and flatten-map/*cmap* as known from functional programming languages. Those are the two operators which are needed to define the MR operator. Our approach is similar to Fegaras's, but in contrast we have one language for both query specification and query execution. An *MRQL* program is first rewritten to a simpler form if possible, and then an algebraic plan is constructed. The operators in such plan are *cmap*s, *groupBy*s and *join*s. Possible optimizations are:
- combining *JOIN*s and *GROUP BY*s on the same key into a single operation, (1) choosing an optimal *JOIN* strategy depending on the data, (2) fusing cascading *cmap*s, (3) fusing *cmaps* with *joins*, (4) synthesizing the *Combine* function. In section 7 we show all those optimizations, except the last one, can be done in NRC. The last one is skipped because for the sake of simplicity we do not include *Combine* functions in our execution model.

## 6 Defining standard operators in NRC

In this section we take a closer look on the operators found in the higher-level MR languages described in the previous section. We show how operators from those three languages can be defined in NRC. This illustrates that our framework generalizes the three considered languages.

### 6.1 SQL operators

We start from the standard SQL operators, which form the basis of the three analyzed languages. For the sake of clarity, we sometimes abuse the notation, to make things clearer, e.g. we avoid the *key-value* pair format in MR expressions, if the keys are not used in the computation. In this section we assume that $x$ is an element from collection $X$, wherever $X$ denotes a collection.

The first and the most basic operator is the projection. Assuming that $X$ has the type $\{\alpha\}$, $\alpha = \langle\cdots\rangle$, and $\pi$ is some function, usually a projection on a subtuple of $\alpha$, we have the following equivalent formulas:

$$\begin{aligned}
\textbf{SQL, MRQL}: &\ \mathit{SELECT}\ \pi(x)\ \mathit{FROM}\ X, \\
\textbf{Pig Latin}: &\ \mathit{FOREACH}\ X\ \mathit{GENERATE}\ \pi(x) \\
\textbf{NRC}: &\ \{\pi(x) \mid x \in X\} \\
\textbf{MR-NRC}: &\ \textbf{MR}\ [\lambda x.\{\pi(x)\}][\lambda x.\textbf{id}_R](X),
\end{aligned}$$

Here $\textbf{id}_R = \{\langle x.k, y\rangle \mid y \in x.vs\}$ and is an “identity” reducer.

The second operator is filtering. Assuming that $X$ is a collection of type $\{\alpha\}$ and $\varphi : \alpha \to \mathbf{boolean}$, the formulas for filtering are as follows:

$$\begin{aligned}
\textbf{SQL, MRQL}: &\ \textit{SELECT} * \textit{FROM } X \textit{ WHERE } \varphi(x),\\
\textbf{Pig Latin}: &\ \textit{FILTER } X \textit{ BY } \varphi(x),\\
\textbf{NRC}: &\ \{x \mid x \in X, \varphi(x)\},\\
\textbf{MR-NRC}: &\ \mathbf{MR}\ [\lambda x.\{y \mid y \in \{x\}, \varphi(y)\}][\lambda x.\mathbf{id}_R](X),
\end{aligned}$$

In some cases it is more efficient to apply projection or filtering in the *Reduce phase*. Corresponding alternative MR versions for these cases are $\mathbf{MR}\ [\lambda x.\{x\}][\lambda x.\{\pi(y) \mid y \in x.vs\}](X)$ and $\mathbf{MR}\ [\lambda x.\{x\}][\lambda x.\{y \mid y \in x.vs, \varphi(y)\}](X)$, respectively. Note that moving those operators, as well as the *cmap*, between the mapper and the reducer is a straightforward rewrite rule.

The third and most complex construct we are interested in is *GROUP BY*. Below we assume that $X$ has the type $\{\alpha\}$, with $\alpha = \langle \dots, \kappa : \beta, \dots \rangle$ and $\pi_\kappa$ is a projection of type $\alpha \to \beta$.

$$\begin{aligned}
\textbf{SQL, MRQL}: &\ \textit{SELECT} * \textit{FROM } X \textit{ GROUP\_BY } \pi_\kappa(x),\\
\textbf{Pig Latin}: &\ \textit{GROUP } X \textit{ BY } \pi_\kappa(x),\\
\textbf{NRC}: &\ \{\{x \mid x \in X, \pi_\kappa(x) \approx y\} \mid y \in \mathbf{set}(\{\pi_\kappa(x) \mid x \in X\})\},\\
\textbf{MR-NRC}: &\ \mathbf{MR}\ [\lambda x.\{\langle k : \pi_\kappa(x), v : x\rangle\}][\lambda x.x](X).
\end{aligned}$$

## 6.2 Pig Latin, HiveQL and MRQL operators

We move to operators unique to higher-level MR languages, viz., *cmap* from the physical layer of MRQL and similar to the comprehension operator, which is based on a *map* construct instead of a *cmap*; and the *COGROUP* from Pig Latin, which can be seen as a generalization of *GROUP* and *JOIN* operators. HiveQL does not add new operators on top of the SQL ones. We leave for future work all forms of the *ORDER BY* operator.

First let us look at the $\textit{cmap}(f)X$ from MRQL, which is based on the concat map well known from the functional languages. The typing of this construct is as follows: $X : \{\alpha\}$, $f : \alpha \to \{\beta\}$ and $\textit{cmap}(f) : \{\alpha\} \to \{\beta\}$. It can be easily expressed in NRC as $\{y \mid x \in X, y \in f(x)\}$.

Provided that $f$ does not change the key, i.e., $f : \langle k : \alpha, v : \beta\rangle \to \langle k : \alpha, v : \gamma\rangle$ such that $f(\langle k : e, v : e'\rangle).k \equiv e$, we can move the application of $f$ between the mapper and the reducer. The efficiency of either choice depends on whether $f$ inflates or deflates the data. In the first case it is better to have it in the reducer, in the second case in the mapper:

$$\begin{aligned}
&\mathbf{MR}\ [\lambda x.f(x)][\lambda x.\mathbf{id}_R](D) \equiv\\
\mathbf{MR}\ [\lambda x.\{x\}][\lambda x.\{z \mid y \in x.vs, z \in f(\langle k : x.k, v : y\rangle)\}](D)&
\end{aligned}$$

This rule can be generalized such that it allows $f$ to be split into a part that is executed in the mapper and a part that is executed in the reducer.

The *COGROUP* is the only operator in which the nested data model is crucial. The syntax of *COGROUP* in Pig Latin is:

$$COGROUP\ X\ BY\ \pi_\kappa(x),\ Y\ BY\ \pi_\iota(y),$$

where $X : \{\alpha\}$, $Y : \{\beta\}$, $\pi_\kappa : \alpha \to \gamma$, and $\pi_\iota : \beta \to \gamma$. The NRC expression for computing *COGROUP* has the form:

$$\{\langle a, \{x \mid x \in X, \pi_\kappa(x) \approx a\}, \{x \mid x \in Y, \pi_\iota(x) \approx a\}\rangle \\ \mid a \in \mathbf{set}(\{\pi_\kappa(x) \mid x \in X\} \uplus \{\pi_\iota(x) \mid x \in Y\})\}.$$

The MR routine for computing the *COGROUP* has the form (note the use of multiple mappers):

$$\begin{aligned}\mathbf{MR}\ \ & [\lambda x.\{\langle k : \pi_\kappa(x), v : \chi_1^2(x)\rangle\}, \lambda x.\{\langle k : \pi_\iota(x), v : \chi_2^2(x)\rangle\}] \\ & [\lambda x.\langle k : x.k, v : \langle\{z \mid y \in x.vs, z \in y.v_1\}, \{z \mid y \in x.vs, z \in y.v_2\}\rangle\rangle](X, Y),\end{aligned}$$

where $\chi_i^j(x)$ stands for $\langle v_1 : \emptyset, \dots v_i : \{x\}, \dots v_j : \emptyset\rangle$. Here the mapper creates tuples with two data fields, the first of which corresponds to the first input, and the second to the second input. Mappers put the input data in the appropriate field as a singleton bag and an empty bag in all other fields. The reducer combines those fields into the resulting tuple. The *COGROUP* is the first operator which spans through both the map and the reduce phase. This is the reason why it is important in the Pig query planner.

## 7 NRC optimizations of higher-level operators

In this section we show how optimizations described in [8, 10, 16] can be represented as NRC rewrite rules. We briefly recall the optimizations mentioned in section 5: (1) pruning unnecessary columns from the data, (2) performing map-side *JOIN*s when possible, (3) combining multiple *JOIN*s on the same key, (4) combining *JOIN*s and *GROUP BY*s on the same key, to a single operation, (5) fusing cascading *cmap*s, (6) fusing *cmap*s with *JOIN*s, (7) computing projections and filterings as early as possible in the intervals between *COGROUP*s – fusing projections and filterings with *COGROUP*s. In the order of appearance, we describe the optimizations and present NRC rewrite rules corresponding with each given optimization. By $=$ we denote syntactic equality, while by $\equiv$ we denote semantic equivalence.

Pruning unnecessary columns strongly depends on the type of the given data and expression. Pruning columns can be easily expressed with well-known NRC rewrite rules, and so we will assume we are working with expressions that project unused columns away as soon as possible.

The map-side join (2) is a technique of computing the join in the mapper, when one of the joined datasets is small enough to fit into a single machine's memory, thus its applicability is data-dependent. It is achieved by replacing the standard reduce-side *COGROUP*-based *JOIN* operator by applying the following rewrite rule:

$$\begin{aligned}\mathbf{MR}\ \ & [\lambda x.\{\langle k : \pi_\kappa(x), v : \chi_1^2(x)\rangle\rangle\}, \lambda x.\{\langle k : \pi_\iota(x), \chi_2^2(x)\rangle\}] \\ & [\lambda x.\{\langle k : x.k, v : \theta(y_1, z_1)\rangle \mid y \in x.vs, z \in x.vs, y_1 \in y.v_1, z_1 \in z.v_2\}](X, Y) \\ \equiv\ \ & \mathbf{MR}\ [\lambda x.\{\langle k : \pi_\kappa(x), v : \theta(x, z)\rangle \mid z \in Y, \pi_\iota(z) \approx \pi_\kappa(x)\}][\lambda x.\mathbf{id}_R](X).\end{aligned}$$

The $\theta$ is a convenience notation for merging the data from two tuples into a single tuple. It is easily NRC expressible as long as we know the the tuple types and how to deal with field name collisions.

As was shown in previous paragraph *JOIN*'s result is usually materialized in the reducer, but if one dataset is small enough it could be materialized in the mapper. We refer to the first as the reduce-side *JOIN*, and to the second one as the map-side *JOIN*. In the following we are discussing combining multiple *JOIN*s together, assuming they join on the same key, and we have to consider three cases: combining two map-side *JOIN*s, combining a map-side *JOIN* with a reduce-side *JOIN* and combining two reduce-side *JOIN*s. Note that any computation in the mapper, before creating the intermediate key, can be seen as a preprocessing. Thus combining a map-side *JOIN* with any other join can be treated as adding an additional preprocessing before the actual MR routine. Hence the first two cases are easy.

The last case, namely combining reduce-side *JOIN*s, and also combining *JOIN*s with *GROUP BY*s on the same key, are generalized by combining *COGROUP* operators which we present here. We define a family of rewrite rules, depending on the number of inputs, and show only an example for three inputs, as generalization is simple:

$$\begin{aligned}
&\mathbf{MR}\ [\lambda x.\{\langle k : x.k, v : \chi^2_1(x.v)\rangle\}, \lambda x.\{\langle k : \pi_\iota(x), v : \chi^2_2(x)\rangle\}]\\
&\quad [\lambda x.\langle k : x.k, v : \langle \{z \mid y \in x.vs, z \in y.v_1.1\},\\
&\quad \{z \mid y \in x.vs, z \in y.v_1.2\}, \{z \mid y \in x.vs, z \in y.v_2\}\rangle\rangle](\mathit{inner(X,Y)}, Z)\\
\equiv\ &\mathbf{MR}\ [\lambda x.\{\langle k : \pi_\kappa(x), v : \chi^3_1(x)\rangle\}, \lambda x.\{\langle k : \pi_\iota(x), v : \chi^3_2(x)\rangle\},\\
&\quad \lambda x.\{\langle k : \pi_\varsigma(x), v : \chi^3_3(x)\rangle\}][\lambda x.\langle x.k, \{z \mid y \in x.vs, z \in y.v_1\},\\
&\quad \{z \mid y \in x.vs, z \in y.v_2\}, \{z \mid z \in x.vs, z \in y.v_3\}\rangle](X, Y, Z),
\end{aligned}$$

where *inner* is a *COGROUP*:

$$\begin{aligned}
\mathbf{MR}\ \ &[\lambda x.\{\langle k : \pi_\kappa(x), v : \chi^2_1(x)\rangle\}, \lambda x.\{\langle k : \pi_\iota(x), v : \chi^2_2(x)\rangle\}]\\
&[\lambda x.\langle k : x.k, v : \langle \{z \mid y \in x.vs, z \in y.v_1\}, \{z \mid y \in x.vs, z \in y.v_2\}\rangle\rangle](X, Y).
\end{aligned}$$

Fusing *cmap*s (5) is a plain NRC rewrite rule, as it is roughly the same as fusing the comprehensions:

$$\begin{aligned}
\mathit{cmap}(f)(\mathit{cmap}(g)D) &= \{y \mid x \in \{g(x) \mid x \in D\}, y \in f(x)\}\\
&\equiv \mathit{cmap}(\lambda x.\{z \mid y \in g(x), z \in f(y)\})(D).
\end{aligned}$$

Note that a composition of *cmap*s is also a *cmap*, hence there is actually never a need to more than a single *cmap* between other operators.

We deal with the last two items (6) and (7) together, as projections and filterings are just a special case of the *cmap* operator. There are two cases of possible fusions. Either the *cmap* may be done on an input of *COGROUP* or *JOIN*, or on their output. Both those cases can be easily represented as rewrite rules. We denote the *cmap*s UDF by $f$ and present only the right hands of the rules, as the left hand sides are straightforward pipelinings of MR routines corresponding respectively to the given constructs for *cmap* before and after *COGROUP*:

$$\begin{aligned} ... = \mathbf{MR}\ & [\lambda x.\{\langle k : \pi_\kappa(x), v : \chi^2_1(f(x))\rangle\}, \lambda x.\{\langle k : \pi_\iota(x), v : \chi^2_2(x)\rangle\}] \\ & [\lambda x.\langle k : x.k, v : \langle\{z \mid y \in x.vs, z \in y.v_1\}, \{z \mid y \in x.vs, z \in y.v_2\}\rangle\rangle](X, Y), \\ ... = \mathbf{MR}\ & [\lambda x.\{\langle k : \pi_\kappa(x), v : \chi^2_1(x)\rangle\}, \lambda x.\{\langle k : \pi_\iota(x), v : \chi^2_2(x)\rangle\}] \\ & [\lambda x.f(\langle k : x.k, v : \langle\{z \mid y \in x.vs, z \in y.v_1\}, \{z \mid y \in x.vs, z \in y.v_2\}\rangle\rangle)](X, Y). \end{aligned}$$

## 8 Conclusion

In this paper we have demonstrated that the Nested Relational Calculus is a suitable language to formulate and reason about MR programs for nested data. It is declarative and higher level than MR, but in some ways lower level than MRQL thus allowing a bit more precise refined optimizations. We showed that MR programs can be expressed in NRC when allowed the same class of UDFs. We also showed that the NRC formalism can express all constructs and optimizations found in Hive, Pig Latin and MRQL. Moreover, NRC is suitable both for writing high-level queries and transformations, as well as MR-based physical evaluation plans when extended with the appropriate constructs. This has the benefit of allowing optimization through rewriting essentially the same formalism, which is not the case for any of the former higher-level MR languages.

Our framework allows for a clear representation of MR programs, which is essential for reasoning about particular programs or the framework in general. NRC is a well defined and thoroughly described language, which has the appropriate level of abstraction to specify the class of MR algorithms we want to concentrate on. It is important that this language is well-designed, much smaller and with a much simpler semantics than other languages than were used to describe MR, like Java or Haskell. This is the reason we think that our work can be potentially more effective than [5, 4].

The higher-level goal of this research is to build a query optimization module that takes as input an NRC expression and translates it into an efficient MR program that can be executed on a MapReduce backend. In future work we will therefore investigate further to what extent NRC and NRC-MR allow for meaningful optimizations through rewriting, either heuristically or cost-based. Moreover, we will investigate the problem of deciding which sub-expressions can be usefully mapped to an MR step, and how this mapping should look in order to obtain an efficient query evaluation plan. This will involve investigating which cost-models are effective for the different types of MapReduce backends.

## References

1. Foto N. Afrati, Anish Das Sarma, Semih Salihoglu, and Jeffrey D. Ullman. Vision paper: Towards an understanding of the limits of map-reduce computation. *CoRR*, abs/1204.1754, 2012.

2. Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand Aiyer. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 1071–1080, New York, NY, USA, 2011. ACM.
3. Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theor. Comput. Sci.*, 149(1):3–48, 1995.
4. Michael J. Cafarella and Christopher Ré. Manimal: relational optimization for data-intensive programs. In *Procceedings of the 13th International Workshop on the Web and Databases*, WebDB '10, pages 10:1–10:6, New York, NY, USA, 2010. ACM.
5. Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 363–375, New York, NY, USA, 2010. ACM.
6. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.
7. Jan Van den Bussche and Stijn Vansummeren. Polymorphic type inference for the named nested relational calculus. *ACM Trans. Comput. Log.*, 9(1), 2007.
8. Leonidas Fegaras, Chengkai Li, and Upa Gupta. An optimization framework for map-reduce queries. In *EDBT*, pages 26–37, 2012.
9. Leonidas Fegaras and David Maier. Optimizing object queries using an effective calculus. *ACM Trans. Database Syst.*, 25(4):457–516, 2000.
10. Alan Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan Narayanam, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a highlevel dataflow system on top of mapreduce: The pig experience. *PVLDB*, 2(2):1414–1425, 2009.
11. Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.
12. Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *SODA*, pages 938–948, 2010.
13. Ralf Lämmel. Google's MapReduce Programming Model – Revisited. *Science of Computer Programming*, 70(1):1–30, 2008.
14. Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SPAA*, page 48, 2009.
15. Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming*, 13(4):277–298, 2005.
16. Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - a warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.
17. Jan Van den Bussche, Dirk Van Gucht, and Stijn Vansummeren. A crash course on database queries. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '07, pages 143–154, New York, NY, USA, 2007. ACM.
18. Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992.