Netezza meets MapReduce
Abstractions for data intensive computing

Severin Hacker and Robert J. Simmons and Carsten Varming

December 19, 2007

Abstract

The high-level goal of our project is to better understand parallel data-intensive computing. In particular we explore, compare and evaluate two different systems: the Netezza platform and Google's MapReduce. We examine their programming model and talk about their limitations. We define a new language, called SRC, which is general enough to target both platforms. We have built a prototype compiler that can compile a SRC program to the Netezza platform. Our evaluation is done on a a large data set of spam emails and we show some preliminary results of using our compiler on this data set.

1 Introduction

An increasing demand for computing power is an ever-present reality, and as applications push against the limits of single-core CPUs and hardware I/O, methods for distributed computation become increasingly relevant. In our work, we take a close look at two such approaches to data-intensive computing, the sub-field of data intensive computing where the bottleneck is generally I/O bandwidth and not CPU power.

One approach to the data-intensive computing problem is to use special-purpose hardware to optimize certain kinds of operations. The Netezza platform [Net07] attempts to achieve this by using an array of disks attached to special-purpose hardware, essentially a FPGA and a microprocessor. By doing so, the special-purpose hardware can execute simple computations on the data while it is streaming from the disk.

A different approach is the use of large-scale parallelism. In particular we look at Google's MapReduce. MapReduce runs on large clusters of commodity PCs connected by a local area network. MapReduce has proven to be an effective way to address many problems in data-intensive computations [DG04], [cYDHP07] and [Läm06], and we specifically evaluate Yahoo's Hadoop system, an open-source reimplementation of the MapReduce platform [Pro07].

From a user perspective, the two different approaches look very different. While MapReduce is built on top of a cluster file-system, Netezza is built on top of a database. The programs for MapReduce are written in a high-level programming language (C++ or Java) while the Netezza platform has to be programmed in low-level, hardware-centric C++. For both platforms, the programs are rather long and contain a lot of platform-specific code. We feel that both platforms try to address overlapping sets of problems, and this project represents an attempt to find a common abstraction layer that can target both platforms (and potentially other platforms such as MSR's Dryad [IBY+07]).

Thus, the goal of our project is to close the gap between the Netezza platform and MapReduce. We first discuss the abstraction that we believe lies at the intersection of the power and expressiveness of the Hadoop platform and the Netezza platform; we name this abstraction ModReduce. We propose a new language, SRC, which is easy to program and is general enough to compile to both the Netezza and Hadoop platforms. We present the design of the language and a prototype compiler that can compile SRC to the Netezza, and demonstrate its use in specifying and querying a database.

2 ModReduce: Bridging the gap

The Hadoop and Netezza platforms both perform data-intensive computing by spreading the data-intensive portion of the work across multiple computational units, and so we started with the expectation that the Netezza abstraction would look something like the MapReduce interface. By reconstructing the algorithm in the style of [Läm06], we can show more clearly Netezza's differences and/or limitations.

Lämmel gives the functions that make up MapReduce the following types:

$$\begin{aligned} map &: \alpha \to (\kappa \times \nu)\ \mathbf{list} \\ reduce &: \kappa \to \nu\ \mathbf{list} \to \mathbf{maybe}\ \beta \end{aligned}$$

...where α is the type of the input and β is the result type. Thus, *map* takes a record (of arbitrary type α) to a list of key-value pairs, and *reduce* takes a key and a list of all values associated with that key, and then maybe produces an output (of arbitrary type β).

2.1 Reconstructing the Netezza interface

The Netezza interface requires coding up the functions in an unusual style, but the functions that it requires the user to implement in order to create an *aggregator* (essentially a reducer) can be modeled in Lämmel's style:

$$\begin{aligned} initState &: unit \to \varphi \\ accum &: \alpha \to \varphi \to \varphi \\ merge &: \varphi \to \varphi \to \varphi \\ finalize &: \varphi \to \beta \end{aligned}$$

Here, α, the input type, can be any type or tuple of types, but β, the output type must be a simple SQL-type. In order to specify a reducer in Netezza you must define an initial state (*initState*). We represent the type of these states as φ. The user also defines a function to add an element to a state (*accum*), merge two states (*merge*), and extract an SQL-element from the state ($finalize$). The Netezza documentation suggests that *accum* is run on a single special-purpose processor, and then *merge* is used to combine results from different units. This means that the user cannot control how the *accum* and *merge* functions interact to produce a result.

We want to find a simpler model that would also translate better to the *reduce* function of MapReduce. We first observe that *finalize* is not crucial - our model allows it to be defined, but if it is not defined it is just the identity function.

Then, because we have no control over how different values are merged, or how many times *initState* is called, in order to ensure a well-defined result the user would generally need to ensure that the value produced by *initState* is the "unit" of the function *merge* (in the same way that 0 is the unit of + and 1 is the unit of ×).

Furthermore, if we define a function *inject* from inputs α to states φ, we can **derive** the function $accum(x, y)$ as $merge(inject(x), y)$. This allows us to see *inject* as optional in the same way that *finalize* is - if we don't specify it, we can just treat it as the identity function.

Therefore, a Netezza aggregator is, at its core, a (commutative and associative) binary operator *merge* and a unit *init*. We can specify them by specifying the following four things:

$$\begin{aligned} init &: \varphi \\ merge &: \varphi \to \varphi \to \varphi \\ inject \text{ (optional)} &: \alpha \to \varphi \\ final \text{ (optional)} &: \varphi \to \beta \end{aligned}$$

As before, there is a restriction that β must be a simple SQL-type.

2.2 ModReduce on Netezza

The Netezza aggregator specification does not give a clear indication of where the power of the Netezza lies. For instance, it is not immediately clear how the aggregator description can be used to aggregate all the values associated with a certain key; this functionality can be achieved with the SQL `group by` option. This is a general problem with the Netezza's interface: the PostgreSQL interface to the system does not give the user a clear idea of what Netezza is supposed to do well in the same way the MapReduce abstraction does.

The aggregators can be seen as a restriction on the *reduce* where the whole list is not available for the programmer to inspect.[1] But how to implement some variant of *map*? Because it was not clear how to talk about emitting the **list** of key/value pairs in SQL, we restricted the *map* phase to only emit zero or one key/value pairs: this is the meaning of the type **maybe** $(\kappa \times \nu)$.

$$mod : \alpha \to \textbf{maybe } (\kappa \times \nu)$$

We call the "map" operation *mod* to point out that it is inferior to *map*. Given a function *mod*, we can produce three axillary functions.

mod_{bool}	$: \alpha \to bool$	If *mod* emits something, return true, otherwise return false
mod_{value}	$: \alpha \to \nu$	If *mod* emits (k, v), returns v. Otherwise undefined.
mod_{key}	$: \alpha \to \kappa$	If *mod* emits (k, v), returns k. Otherwise undefined.

[1]Lämmel discusses that MapReduce has a similar restriction when the *combine* is used.

We could then use these three derived functions to write an SQL query, by using the mod_{bool} function as the argument to SQL's `where` and the mod_{key} function as the argument of SQL's `group by`. Specifically, the SQL query we use to implement the ModReduce query is as follows:

`select` $reduce$`(`mod_{value}`(content)) where` mod_{bool}`(content) group by` mod_{key}`(content)`

2.3 ModReduce on MapReduce

The translation from ModReduce to MapReduce is quite simple. The MapReduce *map* function is just the *mod* function, but it returns a zero- or one-element list instead of returning zero or one values.

The MapReduce *reduce* function takes a key and a list of values; it ignores the key, applies the *inject* function (if any) to each element, and then combines all the elements in the list using the *merge* function.[2] The end result would then have the $final$ function (if any) applied to it.

3 The SRC Language

Following Lämmel's description of MapReduce, we have described the ModReduce algorithm in terms of an abstract functional programming language. This section gives an overview of a simple functional language, SRC, that we provide to allow the user to specify the functions that make up a ModReduce query.

The language we designed has certain features and limitations. Most of the limitations are due the fact that this is a class project with a limited time frame.

The language would ideally have at least the following language features:

- Easy integration with C++ and Java. As both implementations (Netezza and Hadoop) are developing platforms, we would like to have an easy and well defined way to call procedures written in the host language (Java or C++) that could adapt to future changes in those interfaces.
- Type inference. The reduction phase may involve a complicated state with a complicated type (for instance, if you are computing a list of the top twenty values), and type inference can considerably ease the job of the programmer in these situations.
- Polymorphism. In order to reuse code and add abstract reasoning, we would like parametric polymorphism.
- Type Classes. To deal with the many special base types in SQL, we would like type classes to allow proper operator overloading. This will for instance allow us to safely use the same symbols for arithmetic operations on all of SQLs many integer types.

Of the list above we have considered, and at least partially implemented, the first two items. The first item is essential to interact with the host language, and the second is very useful when

[2]These two actions are equivalent to the "map" and "fold" functions common in functional programming.

```
extern > : (int,int) -> bool;
extern * : (int,int) -> int;
infix > 30 non;
infix * 40 left;

>>(x,y) =
  if isnull y then true
  else if isnull x then false
  else valof x > valof y;

infix >> 40 non;

penmax_inj(new:option int) = (new,null);

penmax_merge((max1,pmax1),(max2,pmax2)) =
  if max1 >> max2
  then if pmax1 >> max2 then (max1,pmax1) else (max1,max2)
  else if max1 >> pmax2 then (max2,max1) else (max2,pmax2);

penmax_final(a: option int,b: option int) = b;

export reducer penmax = {merge = penmax_merge, final = penmax_final,
                         inject = penmax_inj, init = (null,null) };
```

Figure 1: Example of finding the "penultimate maximum," i.e. the top two elements of a data set.

writing programs. The last two features adds convenience to the language and make reasoning about programs easier, but due to time constraints we have chosen not to implement them. Our implementation also allows the user to specify functions as infix operators, which can be made left, right, or non associative and given arbitrary precedence.

SQL can specify (per table) if a value might be null. To support this, we borrow the "option" type from ML to represent values that may be null. Option types are analogous to the aforementioned **maybe** type. The language comes with special values and functions ("null","isnull", "valof", and "some") that deal with option types, but we omit the details here.

A program in SRC contains function declarations, declarations of "reducers" or aggregators, infix declarations, and declarations of foreign functions. Figure 1 is an example of a program that:

- Depends on two external functions '>' and '*', which are infix.
- Defines a function >> (which is infix and non-associative).
- Defines three functions: penmax_inj, penmax_merge, and penmax_final
- Declares the aggregator penmax, which computes the "penultimate maximum," the second largest value, in a set of data.

```
f(a,b) =
  h(g(a),a) + h(g(a),a) * h(g(a),b);
```

```
f(a,b) =
  let
   g_a = g(a);
   hg_a = h(g_a,a);
  in hg_a + hg_a * h(g_a,b);
```

Figure 2: An example of let-expressions in SRC. The function `f` to the left does not use let-expressions, while the equivalent function `f` to the right does.

A function declaration consists of a name, a pattern and an expression. The expression may refer to variables in the pattern, previously defined names or the name of the function. The pattern can either be a variable or a list of patterns separated by commas, e.g. (x) is a pattern and $((x, y), z)$ is a pattern.

Expressions can be variables, function calls, tuples, integers, strings, conditionals, and let-expressions. Let-expressions are used to increase clarity and avoid re-computation; for example, in Figure 2 a let-expression is used to keep from computing `g(a)` three times.

3.1 Compiling to C

We have implemented a compiler from SRC to a simple subset of C/C++. The Netezza platform offers only limited processor capacity, so we have made some restrictions to ensure that programs can be compiled into C++ programs that don't use the heap. The main restrictions are the restrictions on functions. We have restricted function declarations to the top level and we don't allow higher order functions. This is general enough to capture the SQL environment we are targeting on Netezza, and specific enough that compiled programs don't utilize a heap. Thus there is no need for garbage collection etc, that we usually see in functional languages.

3.2 Compiling to Java

We have not implemented the compiler from SRC to Java that would be necessary to run SRC queries on Hadoop, but doing so would in many ways be easier than on the C++/Netezza backend, as on the Java backend we don't have the same limitations. We could output Java from more-or-less the stack-allocated low-level intermediate language used for outputting C, but a more natural and expressive (and likely more efficient) translation is possible.

The Java Runtime Environment has a garbage collector, and by the use of inner classes we can compile functions into final objects in order to deal with higher order functions and closures. Recursion can be dealt with by use of the 'this' keyword and mutual recursion[3] can be obtained using Bekic's theorem.[4] Hence the main translation into Java will consist of a translation of our types into either simple types or interfaces, a type-indexed translation of declarations into statements, and a translation of expressions into expressions. This works for all the constructs in our language except let-expressions. The problem with let-expressions is that expressions cannot contain general statements in Java, thus we have to find another way to deal with the declaration

[3]If we were to add it to the language.

[4]Carsten made us say this! See G. Winskei, *The Formal Semantics of Programming Languages*, Chapter 10.

list in the let-expression. This problem can be solved with a standard hoisting technique known from functional program compilation.

4 Evaluation

We have implemented, using OCaml [INR07],[5] a compiler for SRC Code to C, and we have implemented a simple system that allows SQL-like queries and ModReduce queries to be run on the Netezza device. For example, if the "penultimate maximum" query in Figure 1 was stored in a file `penmax.src`, we could query the Netezza database as follows (some output has been omitted):

```
$ ./nzsrc <database> <username> <password>
=> load penmax.src
=> select penmax(b) from myints group by a

 GROUPER22 | REDUC22
-----------+---------
         1 |       6
         3 |      92
         2 |      20
(3 rows)

Elapsed time: 0m0.639s
```

The `nzsrc` program handles reading and typechecking SRC code, compiling that code into C++ classes representing functions and aggregators, loading aggregates and functions into Netezza, calling the Netezza SQL interface, and reporting output to the user. At every query, the interface writes and compiles special-purpose programs connected to that query, so that a call to `not(contains(data,"Nigeria"))` would result in compiling a function `func` that is the composition of the two functions `not` and `contains` specialized with the second argument `"Nigeria"`.[6] This was necessary due to an apparent bug with handling user-defined functions calls with constant arguments, such as `contains(data,"Nigeria")`, on the Netezza, but it is also an optimization as we can partially evaluate a function if some arguments are known.

We currently do not fully support output of ModReduce queries, but it would be straightforward to implement queries as described in Section 2. Qualitatively, our implementation is a proof of concept that the Netezza can in fact be given a simple programming language, SRC, and an interface, `nzsrc`, that captures the interesting computational power of the Netezza while abstracting away the annoying platform-specific coding that Netezza requires. That said, it would take quite a bit more work make our alternative interface stable enough to be attractive to a third party, some of which is discussed in future work.

To perform some quantitative analysis, we ran queries on the Netezza using a slightly-processed corpus of spam data from 2005 (4.3 GB when compressed).

[5] We used OCaml in order to give us the option of closely integrating native C code with SRC code using CIL, which provides a suite of tools for parsing and manipulating C [NMRW02], though we did not explore this possibility in the end.

[6] This is why the sample output looks like `GROUPER22` and `REDUC22` instead of `A` and `PENMAX`.

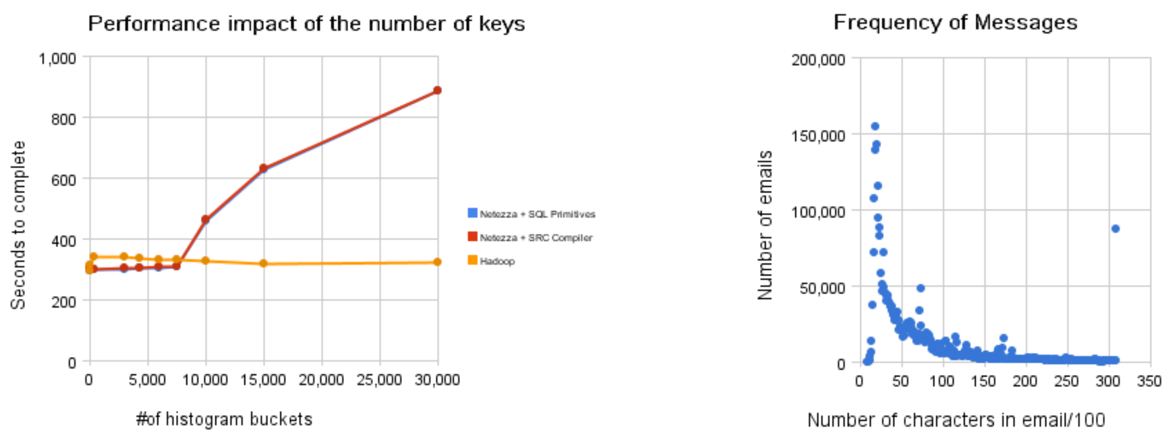

Figure 3: Creating histograms of spam lengths on 2005 spam data. Left: The performance impact of varying the bucket size, and hence the number of buckets. The Hadoop queries ran on 7 nodes and 14 cores, and the Netezza queries ran on a device with 4 hard drive/mini-processor units. Right: The histogram with 301 buckets. The outlier to the right results from truncating long emails.

4.1 Test: SQL built-in versus user-defined

The first tests we wrote were to ascertain whether there was a performance difference between using built-in SQL functions and aggregators versus using hand-coded C++ versions of those functions and aggregators. We tested this by issuing an SQL command to find the maximum length of the "content" field of the spam table using both user-defined and hand-written versions of the "length" function and "max" aggregator. Of the four possible variations, all finished in about five minutes. There was less than a two second difference between the fastest and slowest result, which convinced us that there was essentially no inherent advantage to using a SQL function or aggregate versus a user-defined function or aggregate on the Netezza.

4.2 Micro-benchmark: spam size histogram

In order to determine the performance impact of using the SRC interface, and also to provide a comparison with Hadoop, we issued a series of queries that computed histograms of the spam data at various resolutions. The performance results, and a sample histogram for 100 byte buckets, is shown in Figure 3.

The two Netezza lines in the benchmark almost exactly coincide - the hand-coded examples perform about 5 seconds faster than the SRC examples in most cases, but over the course of a 5-10 minute query this is negligible. Additionally, this graph only counts the query evaluation time; the SRC examples incurred a constant performance hit of an additional few seconds as a result of calling the compiler; again, this is orders of magnitude smaller than the query evaluation time.

Furthermore, the effect of using SRC is dwarfed by the effects of using Netezza at all. The Netezza outperforms Hadoop with a reasonable histogram resolution of $< 7,000$ buckets, performing roughly as well with a fraction of the cores. However, as the histogram buckets get very small, the Netezza loses the ability to efficiently implement the query, possibly because the independent aggregators outstrip the limited memory available on the lightweight Netezza processors.

The lack of performance degradation is not terribly surprising; SRC programs, when compiled, look similar to hand-written programs except that they have a single return point and allocate lots of redundant space on the stack. The C compiler should (and apparently can) optimize these artifacts away, which is one reason we did not optimize this within our compiler. Additionally, much of the costs of type-safe languages involve garbage collection and array bounds checks; since we don't require garbage collection and don't even have arrays in our language, this is not an issue; all string operations, for instance, happen in the "unsafe" native interface.

5 Conclusions and future work

We have given a reconstruction of Netezza's aggregators and described a class of queries that can be run effectively across the Netezza platform and platforms, such as Hadoop, that implement MapReduce. We have implemented a simple programming language for specifying these queries, shown that it can be compiled to Netezza and used as the basis of a greatly simplified interface for Netezza, and argued that providing the same interface to Java/Hadoop is possible. An obvious direction is implementing the interface on other systems, such as Hadoop and Dryad [IBY$^+$07].

Another obvious direction for future work is extending SRC with additional fundamental features as described in Section 3. In order to be generally useful, SRC needs a number of additional features. Basic aggregators such as `count`, `mean`, `median`, etc. should be provided so that programmers need not constantly re-implement such basic functions. It would be useful to add arrays and/or datatypes to the language, and more general support of strings would decrease the need to declare such functions `extern` and write them as C or Java functions.

5.1 SQL compilation and extending MapReduce

However, there is a wealth of less obvious future work suggested by our investigation of the combined expressiveness of two interfaces to two rather different systems, Netezza and Hadoop. As an example, in addition to implementing ModReduce on top of Netezza's SQL interface as described in Section 2.2, we can already support extremely limited SQL queries of this following form on a ModReduce platform, which in turn means they could be run on a system like Hadoop.

`select` f`(x) from myTable where` g`(y) group by` h`(z)`

Here, the *reduce* function is precisely the aggregator f, and *mod* can be easily defined in SRC:

```
mod(x,y,z) = if g(y) then some(h(z),x) else null
```

Unfortunately, this mapping from SQL to ModReduce immediately reveals a rather severe limitation. Implementing a larger subset of SQL `select` queries would almost immediately require operators (joins, etc.) in relational algebra that require the combination of two different relations/tables. However, ModReduce, as we have it outlined in our work, can only access a single table, a limitation that our system shares with MapReduce and Sawzall [PDGQ05].

It seems that part of the expressiveness of joins could be implemented by adding a third phase to our system that matched the returned keys of one query with returned keys of other queries; by then "lining up" the returned data we could use a function called *zip* that merges the two data sets together, a rough analogy to Haskell's *zipWith* function:

```
> zipWith (+) [1,4,6] [2,2,5]
[3,6,11]
```

There is recent previous work on this that we discovered after developing this idea independently [cYDHP07]. In that work, the function, called *merge*, that does more or less what *zip* does; furthermore, they consider sorting properties of SQL and MapReduce that are absent from our specification of ModReduce. They also show how a complex SQL query can be mapped to a MapReduceMerge tree, though they do this by hand, citing automatic translation as future work.

It is clear that neither ModReduce, MapReduce, or MapReduceMerge are the final answer to complex queries on large data sets; they are interfaces that provide a loose guarantee: if you can match your query to our interface, the underlying platform will implement it efficiently. On the other hand are systems such as Dryad [IBY+07] that subsume both Google's MapReduce and relational algebra; the cost is that database queries are done by specifying complex directed graphs, and it is not straightforward to come up with a good Dryad graph. Again, there is no SQL compiler to Dryad; for the time being, mapping these new paradigms back to well-worn query languages like SQL is another interesting question that has not been answered convincingly.

References

[cYDHP07] Hung chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker, *Map-reduce-merge: simplified relational data processing on large clusters*, SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data (New York, NY, USA), ACM, 2007, pp. 1029–1040.

[DG04] Jeffrey Dean and Sanjay Ghemawat, *Mapreduce: simplified data processing on large clusters*, OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation (Berkeley, CA, USA), USENIX Association, 2004, pp. 10–10.

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly, *Dryad: distributed data-parallel programs from sequential building blocks*, EuroSys '07: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (New York, NY, USA), ACM, 2007, pp. 59–72.

[INR07] INRIA, *Ocaml home page*, `http://caml.inria.fr/ocaml/`, 2007.

[Läm06] Ralf Lämmel, *Google's MapReduce Programming Model – Revisited*, Accepted for publication in the Science of Computer Programming Journal; Online since 2 January, 2006; 42 pages, 2006.

[Net07] Netezza, *Netezza home page*, `http://www.netezza.com`, 2007.

[NMRW02] George. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer, *Cil: Intermediate language and tools for analysis and transformation of c programs*, CC '02: Proceedings of the 11th International Conference on Compiler Construction (London, UK), Springer-Verlag, 2002, pp. 213–228.

[PDGQ05] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan, *Interpreting the data: Parallel analysis with sawzall*, Sci. Program. **13** (2005), no. 4, 277–298.

[Pro07] The Apache Project, *Hadoop home page*, `http://lucene.apache.org/hadoop/`, 2007.