Online Appendix for the paper

Efficient Computation of the Well-Founded Semantics over Big Data

Ilias Tachmazidis, Grigoris Antoniou and Wolfgang Faber
University of Huddersfield, UK
(*e-mail:* {ilias.tachmazidis,g.antoniou,w.faber}@hud.ac.uk)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Appendix A MapReduce algorithms

In the appendix, we include the algorithms that are used in the running examples of the paper. More specifically, Algorithm 1 refers to the wordcount example in Section 2.1.

Algorithm 1 Wordcount example

```
map(Long key, String value):                    ▷ key: position in document
for all word $w \in$ value do                   ▷ value: document line
    emit(w, “1”);
end for

reduce(String key, Iterator values):            ▷ key: a word
int count = 0;                                  ▷ values: list of counts
for all value $\in$ values do
    count += parseInt(value);
end for
emit(key, count);
```

In Section 3.1 we described the calculation of the positive goal by applying a single join following Algorithm 2.

Although we mentioned, in Section 3.1, that duplicate elimination should take place as soon as possible in order to minimize overhead, the description of the algorithm was deferred to this appendix. Duplicate elimination can be performed as described in Algorithm 3. Practically, the *Map* function emits every inferred literal as the key, with an empty value. The MapReduce framework performs grouping/sorting resulting in one group (of duplicates) for each unique literal. Each group of duplicates consists of the unique literal as the key and a set of empty values (with values being eventually ignored). The actual duplicate elimination takes place during the reduce phase since for each group of duplicates, we emit the (unique) inferred literal once, using the key, while ignoring the values.

Finally, the calculation of the final goal as described in Section 3.2 follows Algorithm 4.

Algorithm 2 Single join

```
map(Long key, String value):                         ▷ key: position in document (irrelevant)
if value.predicate == "a" then                       ▷ value: document line (literal in $I$)
    emit(value.Z,{value.predicate,value.X});
else if value.predicate == "b" then
    emit(value.Z,{value.predicate,value.Y});
end if

reduce(String key, Iterator values):                 ▷ key: matching argument
List a_List = $\emptyset$, b_List = $\emptyset$;     ▷ values: literals in $I$ for matching
for all value $\in$ values do
    if value.predicate == "a" then
        a_List.add(value.X);
    else if value.predicate == "b" then
        b_List.add(value.Y);
    end if
end for
for all a $\in$ a_List do
    for all b $\in$ b_List do
        emit("ab(a.X,key.Z,b.Y)","");
    end for
end for
```

Algorithm 3 Duplicate elimination

```
map(Long key, String value):                         ▷ key: position in document (irrelevant)
emit(value, "");                                     ▷ value: document line (inferred literal)

reduce(String key, Iterator values):                 ▷ key: inferred literal
emit(key, "");                                       ▷ values: empty values (not used)
```

Algorithm 4 Anti-join

```
map(Long key, String value):                          ▷ key: position in document (irrelevant)
if value.predicate == “ab” then                       ▷ value: document line (literal)
    emit({value.X,value.Z},{value.predicate,value.Y});
else if value.predicate == “c” then
    emit({value.X,value.Z},value.predicate);
end if

reduce(String key, Iterator values):                  ▷ key: matching argument
List ab_List = $\emptyset$;                            ▷ values: literals for matching
for all value $\in$ values do
    if value.predicate == “ab” then
        ab_List.add(value.Y);
    else if value.predicate == “c” then
        return ;                                      ▷ matched by predicate c
    end if
end for
for all ab $\in$ ab_List do
    emit(“abc(key.X,key.Z,ab.Y)”,“”);
end for
```