Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing)

Jens Dittrich[⋆] Jorge-Arnulfo Quiané-Ruiz[⋆] Alekh Jindal[⋆,⋄]
Yagiz Kargin[⋄] Vinay Setty[⋄] Jörg Schad[⋆]

[⋆]Information Systems Group, Saarland University
http://infosys.cs.uni-saarland.de
[⋄]International Max Planck Research School for Computer Science
http://www.imprs-cs.de/

ABSTRACT

MapReduce is a computing paradigm that has gained a lot of attention in recent years from industry and research. Unlike parallel DBMSs, MapReduce allows non-expert users to run complex analytical tasks over very large data sets on very large clusters and clouds. However, this comes at a price: MapReduce processes tasks in a scan-oriented fashion. Hence, the performance of Hadoop — an open-source implementation of MapReduce — often does not match the one of a well-configured parallel DBMS. In this paper we propose a new type of system named Hadoop++: it boosts task performance without changing the Hadoop framework at all (Hadoop does not even 'notice it'). To reach this goal, rather than changing a working system (Hadoop), we *inject* our technology at the right places through UDFs only and affect Hadoop *from inside*. This has three important consequences: First, Hadoop++ significantly outperforms Hadoop. Second, any future changes of Hadoop may directly be used with Hadoop++ without rewriting any glue code. Third, Hadoop++ does not need to change the Hadoop interface. Our experiments show the superiority of Hadoop++ over both Hadoop and HadoopDB for tasks related to indexing and join processing.

1. INTRODUCTION

1.1 Background

Over the past three years MapReduce has attained considerable interest from both the database and systems research community [7, 13, 23, 15, 16, 12, 3, 20, 8, 19, 22, 14, 4, 9, 6].

There is an ongoing debate on the advantages and disadvantages of MapReduce versus parallel DBMSs [1, 11]. Especially, the slow task execution times of MapReduce are frequently criticized. For instance, [16] showed that shared-nothing DBMSs outperform MapReduce by a large factor in a variety of tasks.

Recently, some DBMS vendors have started to integrate MapReduce front-ends into their systems including Aster, Greenplum, and Vertica. However, these systems do not change the underlying execution system: they simply provide a MapReduce front-end to a DBMS. Thus these systems are still databases. The same holds for a recent proposal from VLDB 2009 [3]: *HadoopDB*. It combines techniques from DBMSs, Hive [20], and Hadoop. In summary, HadoopDB can be viewed as a data distribution framework to combine local DBMSs to form a *shared-nothing DBMS*. The results in [3] however show that HadoopDB improves task processing times of Hadoop by a large factor to match the ones of a shared-nothing DBMS.

Proceedings of the VLDB Endowment, Vol. 3, No. 1

1.2 Research Challenge

The approach followed by HadoopDB has severe drawbacks. First, it forces users to use DBMSs. Installing and configuring a parallel DBMS, however, is a complex process and a reason why users moved away from DBMS in the first place [16]. Second, HadoopDB changes the interface to SQL. Again, one of the reasons of the popularity of MapReduce/Hadoop is the simplicity of its programming model. This is not true for HadoopDB. In fact, HadoopDB can be viewed as just another parallel DBMS. Third, HadoopDB locally uses ACID-compliant DBMS engines. However, only the indexing and join processing techniques of the local DBMSs are useful for read-only, MapReduce-style analysis. Fourth, HadoopDB requires deep changes to glue together the Hadoop and Hive frameworks. For instance, in HadoopDB local stores are replaced by local DBMSs. Furthermore, these DBMSs are created outside Hadoop's distributed file system thus superseding the distribution mechanism of Hadoop. We believe that managing these changes is non-trivial if any of the underlying Hadoop or Hive changes[1].

Consequently, the research challenge we tackle in this paper is as follows: is it possible to build a system that: (1) keeps the interface of MapReduce/Hadoop, (2) approaches parallel DBMSs in performance, and (3) does not change the underlying Hadoop framework?

1.3 Hadoop++

Overview. Our solution to this problem is a new type of system: *Hadoop++*. We show that in terms of query processing Hadoop++ matches and sometimes improves the query runtimes of HadoopDB. The beauty of our approach is that we achieve this *without changing the underlying Hadoop framework at all*, i.e. without using a SQL interface and without using local DBMSs as underlying engines. We believe that this non-intrusive approach

[1]A simple example of this was the upgrade from Hadoop 0.19 to 0.20 which affected principal Hadoop APIs.

fits well with the simplicity philosophy of Hadoop.

Hadoop++ changes the internal layout of a *split* — a large horizontal partition of the data — and/or feeds Hadoop with appropriate UDFs. However, Hadoop++ does not change anything in the Hadoop framework.

Contributions. In this paper we make the following contributions:

(1.) The Hadoop Plan. We demonstrate that Hadoop is nothing but a hard-coded, operator-free, physical query execution plan where ten *User Defined Functions* `block`, `split`, `itemize`, `mem`, `map`, `sh`, `cmp`, `grp`, `combine`, and `reduce` are injected at predetermined places. We make Hadoop's hard-coded query processing pipeline explicit and represent it as a DB-style physical query execution plan (*The Hadoop Plan*). As a consequence, we are then able to reason on that plan. (Section 2)

(2.) Trojan Index. We provide a non-invasive, DBMS-independent indexing technique coined *Trojan Index*. A Trojan Index enriches logical input splits by bulkloaded read-optimized indexes. Trojan Indexes are created at data load time and thus have no penalty at query time. Notice, that in contrast to HadoopDB we neither change nor replace the Hadoop framework *at all* to integrate our index, i.e. the Hadoop framework is not aware of the Trojan Index. We achieve this by providing appropriate UDFs. (Section 3)

(3.) Trojan Join. We provide a non-invasive, DBMS-independent join technique coined *Trojan Join*. Trojan join allows us to co-partition the data at data load time. Similarly to Trojan Indexes, Trojan Joins do neither require a DBMS nor SQL to do so. Trojan Index and Trojan Join may be combined to create arbitrarily indexed and co-partitioned data inside the same split. (Section 4)

(4.) Experimental comparison. To provide a fair experimental comparison, we implemented all Trojan-techniques on top of Hadoop and coin the result Hadoop++. We benchmark Hadoop++ against Hadoop as well as HadoopDB as proposed at VLDB 2009 [3]. As in [3] we used the benchmark from SIGMOD 2009 [16]. All experiments are run on Amazon's EC2 Cloud. Our results confirm that Hadoop++ outperforms Hadoop and even HadoopDB for index and join-based tasks. (Section 5)

In addition, Appendix A illustrates processing strategies systems for analytical data processing. Appendix B shows that MapReduce and DBMS have the same expressiveness, i.e. any MapReduce task may be run on a DBMS and vice-versa. Appendices C to E contain additional details and results of our experimental study.

2. HADOOP AS A PHYSICAL QUERY EXECUTION PLAN

In this section we examine how Hadoop computes a MapReduce task. We have analyzed Yahoo!'s Hadoop version 0.19. Note that Hadoop uses a *hard-coded* execution pipeline. No operator-model is used. However Hadoop's query execution strategy may be expressed as a physical operator DAG. To our knowledge, this paper is the first to do so in that detail and we term it *The Hadoop Plan*. Based on this we then reason on The Hadoop Plan.

2.1 The Hadoop Plan

The Hadoop Plan is shaped by three user-defined parameters M, R, and P setting the number of mappers, reducers, and data nodes, respectively [10]. An example for a plan with four mappers ($M = 4$), two reducers ($R = 2$), and four data nodes ($P = 4$) is shown in Figure 1. We observe, that The Hadoop Plan consists of a subplan L (■) and P subplans H1–H4 (■) which correspond to the inital *load phase (HDFS)* into Hadoop's distributed file system. M determines the number of mapper subplans (■), whereas R determines the number of reducer subplans (■).

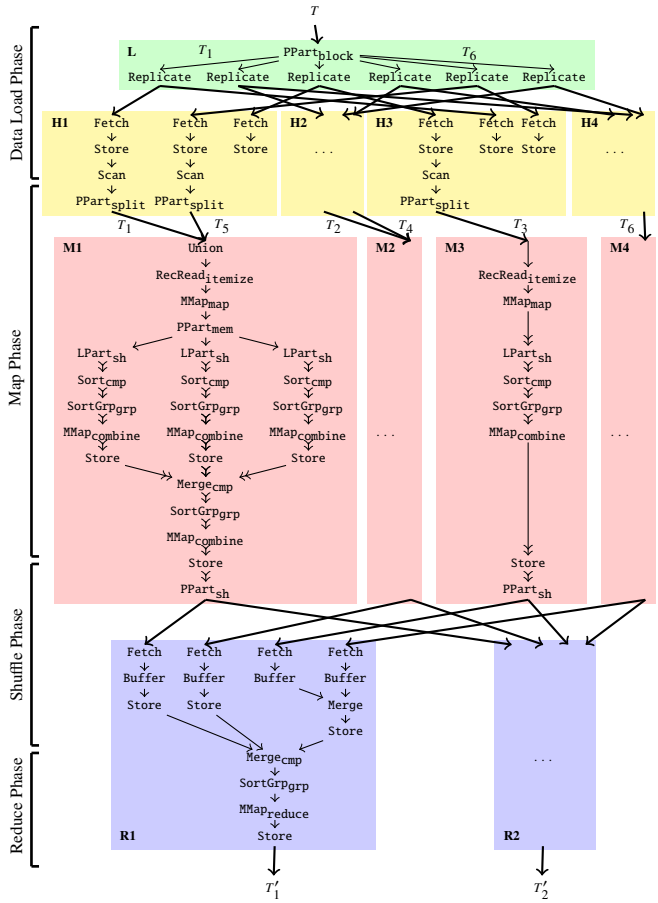


Figure 1: The Hadoop Plan: Hadoop's processing pipeline expressed as a physical query execution plan

Let's analyze The Hadoop Plan in more detail:

Data Load Phase. To be able to run a MapReduce job, we first load the data into the distributed file system. This is done by partitioning the input T horizontally into disjoint subsets $T_1, \ldots, T_b$. See the physical partitioning operator `PPart` in subplan L. In the example $b = 6$, i.e. we obtain subsets $T_1, \ldots, T_6$. These subsets are called *blocks*. The partitioning function `block` partitions the input T based on the block size. Each block is then replicated (`Replicate`). The default number of replicas used by Hadoop is 3, but this may be configured. For presentation reasons, in the example we replicate each block only once. The figure shows 4 different data nodes with subplans H1–H4. Replicas are stored on different nodes in the network (`Fetch` and `Store`). Hadoop tries to store replicas of the same block on different nodes.

Map Phase. In the map phase each map subplan M1–M4 reads a subset of the data called a *split*[2] from HDFS. A *split* is a logical concept typically comprising one or more blocks. This assignment is defined by UDF `split`. In the example, the split assigned to M1 consists of two blocks which may both be retrieved from subplan H1. Subplan M1 unions the input blocks T_1 and T_5 and breaks them into records (`RecRead`). The latter operator uses a UDF `itemize` that defines how a split is divided into items. Then subplan M1 calls `map` on each item and passes the output to a `PPart` operator. This operator divides the output into so-called *spills* based

[2]Not to be confused with *spills*. See below. We use the terminology introduced in [10].

on a partitioning UDF `mem`. By default `mem` creates spills of size 80% of the available main memory. Each spill is logically partitioned (`LPart`) into different regions containing data belonging to different reducers. For each tuple a shuffle UDF `sh` determines its reducer[3]. We use ↠ to visualize the logically partitioned stream. In the example — as we have only two reducers — the stream is partitioned into two substreams only. Each logical partition is then sorted (`Sort`) respecting the sort order defined by UDF `cmp`. After that the data is grouped (`SortGrp`) building groups as defined by UDF `grp`. For each group `MMap`[4] calls UDF `combine` which pre-reduces the data [10]. The output is materialized on disk (`Store`). M1 shows a subplan processing three spill files. These spill files are then retrieved from disk and merged (`Merge`). Again, we apply `SortGrp`, `MMap` and `combine`. The result is stored back on disk (`Store`). Subplan M3 shows a variant where only a single spill file that fits into main memory is created. When compared to M1, M3 looks different. This asymmetry may occur if a mapper subplan (here: M3) consumes less input data and/or creates less output data than other subplans (here: M1). In this case all intermediate data may be kept in main memory in that subplan. In any case all output data will be completely materialized on local disk (`Store`).

Shuffle Phase. The shuffle phase redistributes data using a partitioning UDF `sh`. This is done as follows: each reducer subplan (R1 and R2 in the example) fetches the data from the mapper subplans, i.e. each reduce subplan has a `Fetch` operator for each mapper subplan. Hence, in this example we have $2 \times 4 = 8$ `Fetch` operators (see for instance R1). For each mapper subplan there is a `PPart` operator with R outgoing arrows →. This means, the streams do not represent logical partitions anymore but are physically partitioned (see for instance M1). The reducer subplans retrieve the input files entirely from the mapper subplans and try to store them in main memory in a `Buffer` before continuing the plan. Note that the retrieval of the input to the reduce phase is *entirely blocking*. If the input data does not fit into main memory, those files will be stored on disk in the reducer subplans. For instance, in R1, the input data from M1 and M2 is buffered on disk, whereas the input from M3 and M4 is directly merged (`Merge`) and then stored. After that the input from M1 and M2 and the merged input from M3 and M4 is read from disk and merged. Note that if the input to a reducer is already locally available at the reducer node, `Fetch` may be skipped. This may only happen if the previous mapper subplan was executed on the same node. Also notice that `PPart` uses the same shuffle UDF `sh` as used inside a mapper subplan.

Reduce Phase. Only after a single output stream can be produced, the actual reduce phase starts. The result of the `Merge` is grouped (`SortGrp`) and for each group `MMap` calls `reduce`. Finally, the result is stored on disk (`Store`). The MapReduce framework does not provide a single result output file but keeps one output file per reducer. Thus the result of MapReduce is the union of those files. Notice that all UDFs are optional except `map`. In case `reduce` was not specified, the reduce and shuffle phases may be skipped.

2.2 Discussion

(1.) In general, by using a hard-coded, operator-free, query-execution pipeline, Hadoop makes it impossible to use other more efficient plans (possibly computed depending on current workload, data distribution, etc.)

(2.) At the mapper side, a full-table scan is used as the only access method on the input data. No index access is provided.

[3]By default MapReduce (and also Hadoop) implement UDF `sh` using a hash partitioning on the intermediate key [10].

[4]A multimap operator `MMap` maps one input item to zero, one, or many output item(s). See Appendix B.6 for details.

(3.) Grouping is implemented by sorting.

(4.) Several `MMap` operators executing `combine()` functions (which usually perform the same as a `reduce()` function [10]) are inserted into the merge tree. This is an implementation of *early duplicate removal and aggregation* [5, 21]. For merges with less than three input spills no early aggregation is performed.

(5.) The Hadoop Plan is highly customizable by exchanging one of the ten UDFs `block`, `split`, `itemize`, `mem`, `map`, `sh`, `cmp`, `grp`, `combine`, and `reduce`.

In summary, one could consider The Hadoop Plan a distributed external merge sort where the run (=spill) generation and first level merge is executed in the mapper subplan. Higher level and final merges are executed in the reducer subplans. The sort operation is mainly performed to be able to do a sort-based grouping — but this *interesting order* may also be exploited for applications bulkloading indexes (e.g. inverted lists or B^+-trees). The initial horizontal partitioning into disjoint, equally-sized subsets resembles the strategy followed by shared-nothing DBMSs: in a first phase, the different subsets can be processed fully independently. In a second phase, intermediate results are horizontally repartitioned among the different reducers and then merged into the final result sets.

3. TROJAN INDEX

The Hadoop Plan as shown in Figure 1 uses a `Scan` operator to read data from disk. Currently, Hadoop does not provide index access due to the lack of a priori knowledge of schema and the MapReduce jobs being executed. In contrast, DBMSs require users to specify the schema; indexes may then be added on demand. However, if we know the schema and the anticipated MapReduce jobs, we may create appropriate indexes in Hadoop as well.

Trojan Index is our solution to integrate indexing capability into Hadoop. The salient features of our approach are as follows:

(1.) *No External Library or Engine*: Trojan Index integrates indexing capability natively into Hadoop without imposing a distributed SQL-query engine on top of it.

(2.) *Non-Invasive*: We do not change the existing Hadoop framework. Our index structure is implemented by providing the right UDFs.

(3.) *Optional Access Path*: Trojan Index provides an optional index access path which can be used for selective MapReduce jobs. The scan access path can still be used for other MapReduce jobs.

(4.) *Seamless Splitting*: Data indexing adds an index overhead (~ 8MB for 1GB of indexed data) for each data split. The new logical split includes the data as well as the index. Our approach takes care of automatically splitting indexed data at *logical* split boundaries. Still data and indexes may be kept in different physical objects, e.g. if the index is not required for a particular task.

(5.) *Partial Index*: Trojan Index need not be built on the entire split; it can be built on any contiguous subset of the split as well. This is helpful when indexing one out of several relations, co-grouped in the same split.

(6.) *Multiple Indexes*: Several Trojan Indexes can be built on the same split. However, only one of them can be the primary index. During query processing, an appropriate index can be chosen for data access.

We illustrate the core idea of Trojan Index in Figure 2. For each split of data (SData T) a covering index (Trojan Index) is built. Additionally, a header (H) is added. It contains indexed data size, index size, first key, last key and number of records. Finally, a split footer (F) is used to identify the split boundary. A user can configure the split size (SData T) while loading the data. We

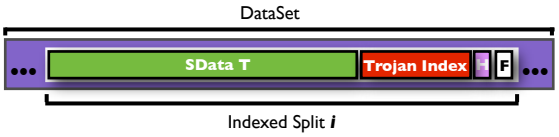


Figure 2: Indexed Data Layout

discuss the Trojan Index creation and subsequent query processing below.

3.1 Index Creation

Trojan Index is a covering index consisting of a sparse directory over the sorted split data. This directory is represented using a cache-conscious CSS-tree [17] with the leaf pointers pointing to pages inside the split. In MapReduce we can express our index creation operation for relation T over an attribute a_i as follows:

INDEX.

$$Index_{a_i}(T) \quad \mapsto \quad \begin{cases} \texttt{map}(\text{key } k, \text{value } v) \mapsto \\ [(\texttt{getSplitID()} \oplus \texttt{prj}_{a_i}(k \oplus v), k \oplus v)] \\ \texttt{reduce}(\text{key } ik, \text{vset } ivs) \mapsto \\ [(ivs \oplus \texttt{indexBuilder}_{a_i}(ivs))] \end{cases}$$

Here, $\texttt{prj}_{a_i}$ denotes a projection to attribute a_i and $\oplus$ denotes that two attribute sets are concatenated to a new schema. Figure 3(a) shows the MapReduce plan corresponding to the indexing operation defined above. The distributed file system stores the data for Relation T. The MapReduce client partitions the Relation T into splits as shown in the figure. The `map` function reads {`offset,record`}-pairs. It emits {`splitID+a,record`} as the intermediate key-value pair. Here `splitID+a` is a composite key concatenating the split ID (function `getSplitID()`) and the index attribute; `record` is a value containing all attributes of the record.

We need to re-partition the composite keys emitted from the mappers such that the reducers receive almost the same amount of data. We do this by supplying a hash partitioning function (UDF `sh` in The Hadoop Plan) that re-partitions records by hashing only on the split identifier portion of the composite key.

$$\texttt{sh}(\text{key } k, \text{value } v, \text{int } numPartitions) \mapsto k.splitID \ \% \ numPartitions \qquad (1)$$

To construct a clustered Trojan Index, the data needs to be sorted on the index attribute a. For this we exploit the interesting orders created by the MapReduce framework [10]. This is faster than performing a local sort at the reducers. To do so, we provide a UDF `cmp` instructing MapReduce to sort records by considering only the second part (the index attribute a) of the composite key.

$$\texttt{cmp}(\text{key } k1, \text{key } k2) \mapsto \texttt{compare}(k1.a\ , k2.a)$$

Since we are building Trojan Index per split, we need to preserve the split in each reducer call. For this we provide a grouping function (UDF `grp`) that groups tuples based on the split identifier portion of the composite key.

$$\texttt{grp}(\text{key } k1, \text{key } k2) \mapsto \texttt{compare}(k1.splitID\ , k2.splitID) \qquad (2)$$

`reduce`, shown in Figure 3(a), has a local `indexBuilder` function. which builds the Trojan Index on the index attribute of the sorted data. `reduce` emits the set of values concatenated with the Trojan Index, index header, and split footer. The output data is stored on the distributed file system.

3.2 Query Processing

Consider a query q referencing an indexed dataset T. We identify the split boundaries using *footer* F and create a map task for each split. Algorithm 1 shows the `split` UDF that we provide for

Algorithm 1: Trojan Index/Trojan Join `split` UDF

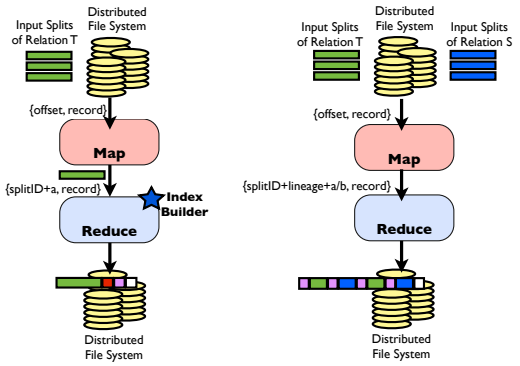
```
   Input  : JobConf job, Int numSplits
   Output: logical data splits
 1 FileSplit [] splits;
 2 File [] files = GetFiles(job);
 3 foreach file in files do
 4 |   Path path = file.getPath();
 5 |   InputStream in = GetInputStream(path);
 6 |   Long offset = file.getLength();
 7 |   while offset > 0 do
 8 |   |   in.seek(offset-FOOTER_SIZE);
 9 |   |   Footer footer = ReadFooter(in);
10 |   |   Long splitSize = footer.getSplitSize();
11 |   |   offset -= (splitSize + FOOTER_SIZE);
12 |   |   BlockLocations blocks = GetBlockLocations(path,offset);
13 |   |   FileSplit newSplit = CreateSplit(path,offset,splitSize,blocks);
14 |   |   splits.add(newSplit);
15 |   end
16 end
17 return splits;
```

Algorithm 2: Trojan Index `itemize`.initialize UDF

```
   Input: FileSplit split, JobConf job
 1 Global FileSplit split = split;
 2 Key lowKey = job.getLowKey();
 3 Global Key highKey = job.getHighKey();
 4 Int splitStart = split.getStart();
 5 Global Int splitEnd = split.getEnd();
 6 Header h = ReadHeader(split);
 7 Overlap type = h.getOverlapType(lowKey,highKey);
 8 Global Int offset;
 9 if type == LEFT_CONTAINED or type == FULL_CONTAINED or type ==
   POINT_CONTAINED then
10 |   Index i = ReadIndex(split);
11 |   offset = splitStart + i.lookup(lowKey);
12 else if type == RIGHT_CONTAINED or type == SPAN then
13 |   offset = splitStart;
14 else
15 |   // NOT_CONTAINED, skip the split;
16 |   offset = splitEnd;
17 end
18 Seek(offset);
```

creating the splits. For a given job, we retrieve and iterate over all data files (Lines 2–3). For each file we retrieve its path and the input stream (Lines 4–5). The input stream is used to seek and read the split footers, i.e. we do not scan the entire data here. We start looking for footers from the end (Lines 6–8) and retrieve the split size from them (Lines 9–10). We set the offset to the beginning of the split (Line 11) and use it to retrieve block locations (Line 12) and to create a logical split (Line 13). We add the newly created split to the list of logical splits (Line 14) and repeat the process until all footers in all files have been read. Finally, we return the list of logical splits (Line 17).

Algorithm 2 shows the `itemize` UDF that we provide for index scan. We read the low and the high selection keys (Lines 1–2) from the job configuration and the split boundary offsets (Lines 3–4) from the split configuration. Thereafter, we first read the index header (Line 5) and evaluate the overlap type (Line 6) i.e. the portion of the split data relevant to the query. Only if the split contains the low key (Line 8), we read the index (Line 9) and compute the low key offset within the split (Line 10). Otherwise, if the split contains the high key or the selection range spans the split (Line 11), we set the offset to the beginning of the split (Line 12); else we skip the split entirely (Lines 13–15). Finally, we seek the offset within the split (Line 17) to start reading data record by record. Algorithm 3 shows the method to get the next record from the data split. We check if the split offset is within the end of split (Line 1) and



(a) Indexing (b) Co-partitioning

Figure 3: MapReduce Plans

Algorithm 3: Trojan Index `itemize`.next UDF

```
   Input  : KeyType key, ValueType value
   Output: has more records
1  if offset < splitEnd then
2      Record nextRecord = ReadNextRecord(split);
3      offset += nextRecord.size();
4      if nextRecord.key < highKey then
5          SetKeyValue(key, value, nextRecord);
6          return true;
7      end
8  end
9  return false;
```

index key value of the next record is less than the high key (Line 3). If yes, we set the key and the value to be fed to the mapper and return true (Lines 4-5), indicating there could be more records. Else, we return false (Line 8).

Note that the use of the Trojan Index is optional and depends upon the query predicate. Thus, both full and index scan are possible over the same data. In addition, indexes and data may be kept in separate physical blocks, i.e. UDF `split` may compose physical blocks into logical splits suited for a particular task.

4. TROJAN JOIN

Efficient join processing is one of the most important features of DBMSs. In MapReduce, two datasets are usually joined using *repartitioning*: partitioning records by join key in the map phase and grouping records with the same key in the reduce phase. The reducer joins the records in each key-based group. This re-partitioned join corresponds to the join detailed in Appendix B.3. Yang et al. [23] proposed to extend MapReduce by a third *Merge* phase. The Merge phase is a join operator which follows the reduce phase and gets sorted results from it. Afrate and Ullman [4] proposed techniques to perform multiway joins in a single MapReduce job. However, all of the above approaches perform the join operation in the reduce phase and hence transfer a large amount of data trough the network — which is a potential bottleneck. Moreover, these approaches do not exploit any schema-knowledge, which is often available in advance for many relational-style tasks. Furthermore, join conditions in a schema are very unlikely to change — the set of tables requested in a join query may however change.

Trojan Join is our solution to support more effective join processing in Hadoop. We assume that we know the schema and the expected workload, similar to DBMS and HadoopDB. The core idea is to *co-partition* the data at load time — i.e. given two input relations, we apply the same partitioning function on the join attributes of both the relations at data loading time — and place the

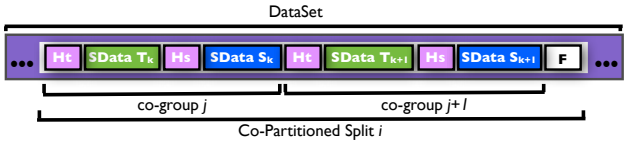


Figure 4: Co-partitioned Data Layout

co-group pairs, having the same join key from the two relations, on the same split and hence on the same node. As a result, joins are now processed locally within each node at query time — a feature that is also explored by SQL-DBMSs. Moreover, we are free to group the data on any attribute other than the join attribute in the same MapReduce job. The salient features of Trojan Join are as follows:

(1.) *Non-Invasive*. We do not change the existing Hadoop framework. We only change the internal representation of a data split.

(2.) *Seamless Splitting*. When co-grouping the data, we create three headers per data split: two for indicating the boundaries of data belonging to different relations; one for indicating the boundaries of the logical split. Trojan Join automatically splits data at logical split boundaries that are opaque to the user.

(3.) *Mapper-side Co-partitioned Join*. Trojan Join allows users to join relations in the map phase itself exploiting *co-partitioned data*. This avoids the shuffle phase, which is typically quite costly from the network traffic perspective.

(4.) *Trojan Index Compatibility*. Trojan indexes may freely be combined with Trojan Joins. We detail this aspect in Section 4.3.

We illustrate the data layout for Trojan Join in Figure 4. Each split is separated by split footer (F) and contains data from two relations T S (depicted green and blue in Figure 4). We use two headers H_t and H_s, one for each relation, to indicate the size of each co-partition[5]. Given an equi-join predicate $PJ(T,S) = (T.a_i = S.b_j)$, the Trojan Join proceeds in two phases: the *data co-partitioning* and *query processing* phases.

4.1 Data Co-Partitioning

Trojan Join co-partitions two relations in order to perform join queries using map tasks only. Formally, we can express co-partitioning as:

$$CoPartition_{a_i,b_j}(T,S) \mapsto$$

$$\begin{cases} \texttt{map}(\text{key } k, \text{value } v) \mapsto \\ \begin{cases} [(\texttt{getSplitID}() \oplus \texttt{T} \oplus \texttt{prj}_{a_i}(k \oplus v), k \oplus v)] & \text{if input}(k \oplus v) = T, \\ [(\texttt{getSplitID}() \oplus \texttt{S} \oplus \texttt{prj}_{b_j}(k \oplus v), k \oplus v)] & \text{if input}(k \oplus v) = S. \end{cases} \\ \texttt{reduce}(\text{key } ik, \text{vset } ivs) \mapsto [(\{ik\} \times ivs)] \end{cases}$$

Here, the helper input() function identifies whether an input record belongs to T or S. Figure 3(b) shows the MapReduce plan for co-partitioning the data. This works as follows. The MapReduce client partitions the data of both relations into splits as shown in the figure. For each record in an input split, `map` receives the `offset` as key and the record as value. It emits {`splitID+lineage+joinvalue, record`} as key-value pairs. Here `splitID+lineage+joinvalue` is a composite key concatenating the ID of the split; the lineage of the record (belongs to T or S); and the value of its join attribute (either a_i or b_j depending on the lineage); `record` contains all attributes of the record.

For re-partitioning and grouping the key-value pairs we use the same `sh` and `grp` UDFs as in index creation (See Equa-

[5]Notice that one can also store each relation in separate physical blocks just like a DBMS. Extending our approach to this is straightforward: we simply need to provide a UDF `split`. This also holds for our Trojan Index proposal.

Algorithm 4: Trojan Join `itemize`.next UDF

```
   Input  : KeyType key, ValueType value
   Output: has more records
1  if offset < splitEnd then
2      if offset == nextHeaderOffset then
3          Header header = ReadHeader(split);
4          offset += header.size();
5          nextHeaderOffset = offset + header.getSplitSize();
6      end
7      Record nextRecord = ReadNextRecord(split);
8      offset += nextRecord.size();
9      SetKeyValue(key, value, nextRecord);
10     return true;
11 end
12 return false;
```

tions 1 and 2). However, we provide the following `cmp` UDF for sorting:

$$\texttt{cmp}(\text{key } k1, \text{key } k2) \mapsto \begin{cases} \text{int } \mathit{lineageCmp} = \texttt{compare}(k1.\mathit{lineage}\ ,\ k2.\mathit{lineage}) \\ \text{if } (\mathit{lineageCmp} == 0)\ \{ \\ \quad \mathit{output}:\ \texttt{compare}(k1.a\ ,\ k2.b) \\ \}\ \text{else}\ \{ \\ \quad \mathit{output}:\ \mathit{lineageCmp} \\ \} \end{cases}$$

As a result, each call to `reduce` receives the set of records having the same join attribute value. The final output of `reduce` is a virtual split containing several co-groups as shown in Figure 4.

4.2 Query Processing

A Trojan Join between relations T and S can be expressed as the re-partitioned join operator shown in Appendix B.3 replacing `map` with an identity function. Though join processing in this manner is a considerable improvement, we still need to shuffle the data. However, we actually do not need the shuffle phase as relations T and S were already co-partitioned. Therefore, we present an optimized variant of this join which requires only a single `map` without a `reduce`. Hence, Hadoop++ may skip both the shuffle and the reduce phase. The map function in Trojan Join is shown below:

$$T \bowtie_{PJ(T,S)} S \mapsto \begin{cases} \mathit{set}\ \beta = \emptyset; \mathit{key}\ lk = \mathit{null}; \\ \texttt{map}(\text{key } k, \text{value } v) \mapsto \\ \begin{cases} \text{if } (lk\ \texttt{!=}\ k)\ \{ \\ \quad \text{if } !\mathit{first\_incoming\_pair}(k, v)\ \{ \\ \quad\quad \mathit{output}: \Big[\texttt{crossproduct}(T'_{lk}, S'_{lk})\ | \\ \quad\quad\ T'_{lk} = \{(\texttt{prj}_{a_i}(k \oplus v), k \oplus v)\ |\ (k \oplus v) \in \beta \wedge \text{if input}(k \oplus v) = T\}, \\ \quad\quad\ S'_{lk} = \{(\texttt{prj}_{b_j}(k \oplus v), k \oplus v)\ |\ (k \oplus v) \in \beta \wedge \text{if input}(k \oplus v) = S\}\Big] \\ \quad \} \\ \quad \beta = \{k \oplus v\},\ \ lk = k \\ \} \\ \text{else}\ \{ \\ \quad \mathit{output}: \text{none} \\ \quad \beta = \beta \cup \{k \oplus v\} \\ \} \end{cases} \end{cases}$$

To process a join query, our approach automatically splits the required data by identifying the split boundaries — using the *footer* F — and creates a map task for each split. For this, we supply a `split` UDF that identifies such boundaries (see Algorithm 1). We also supply a UDF `itemize` that allows mappers to skip headers in input splits. Algorithm 4 shows how UDF `itemize` computes the next key-value pairs ('items'). Here `offset`, `splitEnd`, and `header` are global variables defined in the `itemize`.initialize function (similar to Algorithm 2). We check if the split `offset` is contained in this split (Line 1). If yes, we check if the current `offset` points to a header (Line 2) so as to skip the header (Lines 3–5). We then set the key and the value to be fed to `map` and return true (Lines 7–10), indicating there could be more records. In case the `offset` is not within the end of split, we return false (Line 12).

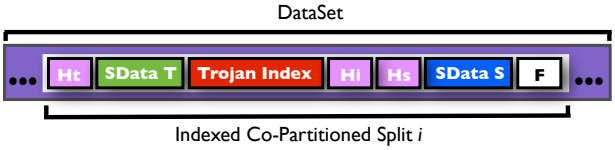


Figure 5: Indexed Co-partitioned Data Layout

This indicates that there are no more records.

The `map` function shown before starts by initializing a co-group with the first (k, v)-pair. Thereafter, it keeps collecting in β the records belonging to the same co-group i.e. the same join attribute values. A different join attribute value indicates the beginning of the next co-group in the data. Here, we make two assumptions: first, records with the same join attribute value arrive contiguously, which is realistic since the relations are co-grouped; second, in contrast to previous MapReduce jobs, the map function maintains a state (β, lk) to identify the co-group boundaries within a split. When a new co-group starts, the map function classifies the records in β into relations T' and S' based on their lineage and performs the cross product between them by calling the local `crossproduct` function. The result is emitted and β is reset to start collecting records for the next co-group. This process is repeated until there is no more incoming (k, v)-pair. To perform the cross product on the last co-group, the `map` injects an `end-of-split record` after the last record in each data split marking the end of that split. The `reduce` may then output the join result over the last co-group. Notice that the final result of all of these co-partitioned joins is exactly the same as the result produced by the re-partitioned join.

4.3 Trojan Index over Co-Partitioned Data

We can also build indexes on co-partitioned data. Trojan Join may be combined with both unclustered and clustered Trojan Indexes. For instance, we can build an unclustered Trojan Index over any attribute without changing the co-grouped data layout. Alternatively, we can build a clustered Trojan Index by internally sorting the co-partitioned data based on the index attribute. The internal sorting process is required only when the index attribute is different from the join attribute. For example, assume relations T and S are co-partitioned and suppose we want to build a clustered Trojan Index over a given attribute of relation T. To achieve this, we run the indexing MapReduce job as described in Section 3.1. This job sorts the records from T based on the index attribute and stores them contiguously within the split. The resulting data layout is illustrated in Figure 5. Each split is separated by a split footer (F) and has a header per relation (H_t and H_s), indicating the size of each co-partition. In addition, a clustered Trojan Index and its header (H_i) is stored after the indexed relation (T) in the split. At query time, we supply the UDF `itemize` function as before. However, we set the constructor of `itemize` function as in Algorithm 3 in order to provide index scan. Adapting Trojan Join processing for indexed data is straightforward.

5. EXPERIMENTS

We evaluate the performance of Hadoop++ (i.e. Hadoop including Trojan Index and Trojan Join) and compare it with Hadoop and HadoopDB. Our main goal in the experiments is to show that we can reach similar or better performance than Hadoop and HadoopDB without relying on local DBMSs. We also show in §5.3 that Hadoop++ still inherits Hadoop's fault-tolerance performance.

5.1 Benchmark Setup

We ran all our experiments on Amazon EC2 large instances in *US-east* location. Each large instance has 4 EC2 compute units (2 virtual cores), 7.5 GB of main memory, 850 GB of disk storage and

runs 64-bit platform Linux Fedora 8 OS. Throughout our performance study we realized that performance on EC2 may vary. We analyse this variance in detail in an accompanying paper [18]. Here we executed each of the tasks three times and report the average of the trials. We discard these assumptions to evaluate fault-tolerance in §5.3. We report only those trial results where all nodes are available and operating correctly. To factor out variance, we also ran the benchmark on a physical 10-node cluster where we obtained comparable results[6]. On EC2 we scale the number of virtual nodes: 10, 50, and 100. We compared the performance of Hadoop++ against Hadoop and HadoopDB. We used Hadoop 0.19.1 running on Java 1.6 for all these three systems. We evaluated two variants of Hadoop++ that only differ in the size of the input splits (256 MB and 1 GB[7]). For HadoopDB, we created databases exactly as in [3]. Appendix D lists configuration details.

We used the benchmark and data generator proposed in [16] and used in the HadoopDB paper [3]. We selected those tasks relevant to indexing and join processing. For completeness, we also report results of the other tasks in Appendix E. The benchmark creates three tables: (1) *Documents* containing HTML documents, each of them having links to other pages following a Zipfian distribution. (2) *Rankings* containing references to *Documents*, (3) *UserVisits* referencing *Rankings*. Both *Rankings* and *UserVisits* contain several randomly generated attribute values. The sizes of *Rankings* and *UserVisits* are 1 GB (18M tuples) and 20 GB (155M tuples) per node, respectively. Please refer to [16] for details.

5.2 Analytical Tasks

5.2.1 Data Loading

As in [3] we show the times for loading UserVisits only; the time to load the small Rankings is negligible. Hadoop just copies *UserVisits* (20GB per node) from local hard disks into HDFS, while Hadoop++ and HadoopDB partition it by *destinationURL* and index it on *visitDate*. Figure 6(a) shows the load times for *UserVisits*. For Hadoop++ we show the different loading phases: The data loading into HDFS including conversion from textual to binary representation, followed by the co-partioning phase (§ 4.1), and index creation (§ 3.1). We observe that Hadoop++(256MB) has similar performance as HadoopDB; Hadoop++(1GB), however, is slightly slower. We believe this is because the loading process is CPU-bound, thereby causing map tasks to slow down when processing large input splits. However, this difference is negligible, as these costs happen at data load time. This means these costs have to be paid only once. Users may then run an unlimited number of tasks against the data. The **trade-off** we observe is similar to the one seen in any DBMS: the more we invest at data load time, the more we might gain at query time. Thus, the more queries benefit from that initial investment, the higher the overall gain. Overall, we conclude that Hadoop++ scales well with the number of nodes.

5.2.2 Selection Task

This task performs a selection predicate on *pageRank* in *Rankings*. We use the same selectivity as in [3, 16], i.e. 36,000 tuples per node by setting the *pageRank* threshold to 10. The SQL queries and MapReduce jobs used for the selection task are described in Appendix C.1. For this task, we run two variants of HadoopDB similar to the authors of HadoopDB [3]. In the first variant, each node contains the entire 1 GB *Rankings* in a single local database. In the second variant each node contains twenty partitions of 50 MB each in separate local databases (HadoopDB Chunks). Figure 6(b) illustrates the selection task results for all systems. We observe that Hadoop++ outperforms Hadoop and HadoopDB Chunks by up to factor 7, and HadoopDB by up to factor 1.5. We also observe that Hadoop++(1GB) performs better than Hadoop++(256MB). This is because Hadoop++(1GB) has much fewer map tasks to execute and hence less scheduling overhead. Furthermore, its index coverage is greater. This allows it to get more data at once. These results demonstrate the superiority of Hadoop++ over the other systems for selection tasks.

5.2.3 Join Task

This task computes the average *pageRank* of those pages visited by the *sourceIP* address that has generated the most *adRevenue* during the week of January 15-22, 2000. This task requires each system to read two different data sets (*Rankings* and *UserVisits*) and join them. The number of records in *UserVisits* that satisfy the selection predicate is ~134,000. The SQL queries and MapReduce jobs used to perform the join tasks are shown in Appendix C.2.

Figure 6(c) illustrates results for each system when performing this join task. Again, we observe that Hadoop++ outperforms Hadoop by up to factor 20. This is because Hadoop++ performs an index-scan over *UserVisits* to speed up the selection predicate and because *Rankings* and *UserVisits* were co-grouped at loading time. More importantly, our results show that Hadoop++(1GB) outperforms HadoopDB by up to factor 1.6. This is not the case for Hadoop++(256MB), because it has less relevant data per input split to join and more `map` tasks to process. Again, as discussed in §5.2.1, these gains are possible, as we trade query performance with additional effort at data load time, see Figure 6(a).

5.3 Fault-Tolerance

In this section we show results of two fault-tolerance experiment which are similar to the one done in [3]. We perform the *node failures* experiment as follows: we set the expiry interval, i.e. the maximum time between two heartbeats, to 60 seconds. We chose a node randomly and kill it after 50% percent of work progress. We perform the *straggler nodes* experiment as follows: we run a concurrent I/O-intensive process on a randomly chosen node so as to make it a straggler node. We define the slowdown as in [3], $slowdown = \frac{(n-f)}{n} * 100$, where n is the query execution time without failures and f is the execution time with a node failure. For both series of tests, we set HDFS replication to 2.

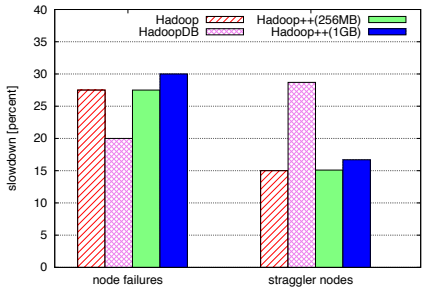
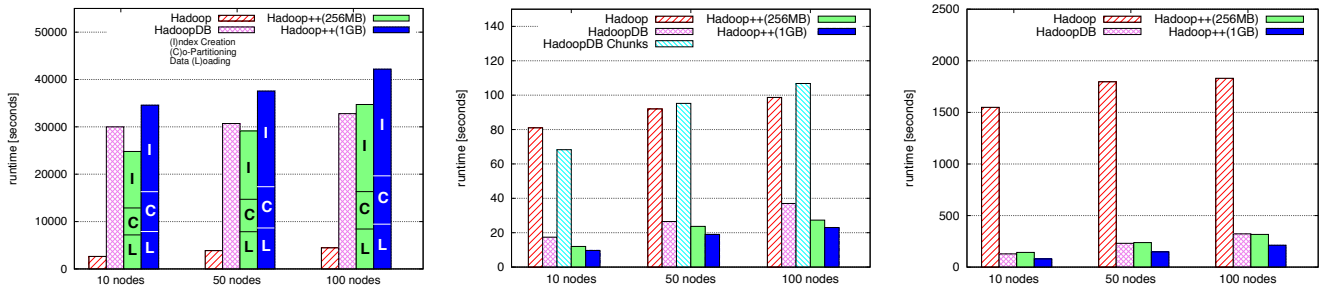


Figure 7: Fault Tolerance.

Figure 7 shows the results. As expected, we observe that Hadoop++(256MB) has the same performance as Hadoop. However, we can see that while increasing the size of input splits from 256 MB to 1 GB, Hadoop++ slows down. This is because Hadoop++(1GB) has 4 times more data to process per input split, and hence it takes more time to finish any lost task. Hence, we ob-

[6]With a single exception: on the physical cluster for the selection task Hadoop++(1GB) was still faster than HadoopDB, but Hadoop++(256MB) was slightly slower than HadoopDB.

[7]Unfortunately, we could not use split sizes beyond 1GB due to a bug [2] in Hadoop's distributed file system. We believe however that runtimes using Trojan Join would improve even further.



(a) Data loading, partitioning, and indexing (b) Selection Task (c) Join Task

Figure 6: Benchmark Results related to Indexing and Join Processing

serve a natural trade-off between performance and fault-tolerance: By increasing the input split size, Hadoop++ has better performance but it is less fault-tolerant and vice-versa. We observe that Hadoop++ is slower than HadoopDB for the node failures experiments. This is because Hadoop++ needs to copy data from replica nodes while HadoopDB pushes work to replica nodes and thus requires less network traffic. For the straggler nodes experiment however, Hadoop++ significantly outperforms HadoopDB. This is because HadoopDB sometimes pushes tasks to straggler nodes rather than replica nodes. This slows down its speculative execution.

6. DISCUSSION & CONCLUSION

This paper has proposed new index and join techniques: Trojan Index and Trojan Join, to improve runtimes of MapReduce jobs. Our techniques are non-invasive, i.e. they do to require us to change the underlying Hadoop framework. We simply need to provide appropriate user-defined functions (and not only the two functions `map` and `reduce`). The beauty of this approach is that we can incorporate such techniques to any Hadoop version with no effort. We exploited this during our experiments when moving from Hadoop 0.20.1 to Hadoop 0.19.0 (used by HadoopDB) for fairness reasons. We implemented our Trojan techniques on top of Hadoop and named the resulting system Hadoop++.

The experimental results demonstrate that Hadoop++ outperforms Hadoop. Furthermore, for tasks related to indexing and join processing Hadoop++ outperforms HadoopDB – without requiring a DBMS or deep changes in Hadoop's execution framework or interface. We also observe that as we increase the split size, Hadoop++ further improves for both selection and join tasks. This is because the index coverage also increases. Performance of fault-tolerance, however, decreases with larger splits as it requires more time to recompute lost tasks. This symbolizes a tradeoff between runtime and fault tolerance of MapReduce jobs.

An important lesson learned from this paper is that most of the performance benefits stem from exploiting schema knowledge on the dataset and anticipating the query workload at *data load time*. Only if this schema knowledge is available, DBMSs, HadoopDB as well as Hadoop++ may improve over Hadoop. But again: there is no need to use a DBMS for this. Schema knowledge and anticipated query workload may be exploited in *any* data processing system.

In terms of Hadoop++'s interface we believe that we do not have to change the programming interface to SQL: standard MapReduce jobs — unaware of possible indexes and join conditions — may be analyzed [6] and then rewritten to use the Trojan techniques proposed in this paper.

Acknowledgments. We would like to thank all students of the MapReduce/PigLatin LAB at UdS (summer'09) for the fruitful discussions. Work partially supported by Saarbrücken Cluster of Excellence MMCI, UdS Graduate School for CS, and IMPRS CS.

7. REFERENCES

[1] Dbcolumn on MapReduce, http://databasecolumn.vertica.com /2008/01/mapreduce-a-major-step-back.html.
[2] HDFS Bug, http://issues.apache.org/jira/browse/HDFS-96.
[3] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB*, 2(1), 2009.
[4] F. Afrati and J. Ullman. Optimizing Joins in a Map-Reduce Environment. In *EDBT*, 2010.
[5] D. Bitton and D. J. DeWitt. Duplicate Record Elimination in Large Data Files. *TODS*, 8(2), 1983.
[6] M. J. Cafarella and C. Re. Relational Optimization for Data-Intensive Programs. In *WebDB*, 2010.
[7] R. Chaiken et al. Scope: Easy and Efficient Parallel Processing of Massive Data Sets. *PVLDB*, 1(2), 2008.
[8] J. Cohen, B. Dolan, M. Dunlap, J. Hellerstein, and C. Welton. Mad Skills: New Analysis Practices for Big Data. *PVLDB*, 2(2), 2009.
[9] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. In *NSDI*, 2010.
[10] J. Dean and S. Ghemawat. Mapreduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.
[11] J. Dean and S. Ghemawat. MapReduce: A Flexible Data Processing Tool. *CACM*, 53(1):72–77, 2010.
[12] A. Gates et al. Building a HighLevel Dataflow System on Top of MapReduce: The Pig Experience. *PVLDB*, 2(2), 2009.
[13] M. Isard et al. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *EuroSys*, 2007.
[14] K. Morton and A. Friesen. KAMD: A Progress Estimator for MapReduce Pipelines. In *ICDE*, 2010.
[15] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD*, 2008.
[16] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, 2009.
[17] J. Rao and K. A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *VLDB*, 1999.
[18] J. Schad, J. Dittrich, and J.-A. Quiane-Ruiz. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *PVLDB*, 3(1), 2010.
[19] M. Stonebraker, D. J. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. MapReduce and Parallel DBMSs: Friends or Foes? *CACM*, 53(1), 2010.
[20] A. Thusoo et al. Hive - a warehousing solution over a map-reduce framework. *PVLDB*, 2(2), 2009.
[21] P. Yan and P. Larson. Data Reduction Through Early Grouping. In *CASCON*, 1994.
[22] C. Yang, C. Yen, C. Tan, and S. Madden. Osprey: Implementing MapReduce-Style Fault Tolerance in a Shared-Nothing Distributed Database. In *ICDE*, 2010.
[23] H. Yang et al. Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In *SIGMOD*, 2007.

APPENDIX

A. EXECUTION STRATEGIES IN THE FOUR SYSTEMS

The goal of this section is to show how the four systems to large scale data analysis Parallel DBMS, MapReduce, Hybrid Approach, and our proposal Hadoop++ process a simple analytical task. As an example, consider we want to build an inverted buzzword search index on a paper collection. The input data consists of unstructured text documents each having a distinct *D*ocument *ID* (DID).

A.1 Parallel DBMS

We first have to define appropriate schemas in the PDBMS using SQL. We need schemas for the input table as well for the final output index. An appropriate schema for the input may be Documents(DID: INT, term:VARCHAR) and InvertedIndex(buzzword:VARCHAR, postingList:VARCHAR). Second, we need to load the input text documents, tokenize them and for each term we create an entry (DID,term) in Documents. Third, we call:

```
SELECT term, buildPostingList(DID) FROM Documents
WHERE isBuzzword(term) GROUP BY term;
```

This means, we only consider the buzzwords from Documents by probing UDF `isBuzzword`, group the results on term, and for each term we create a posting list by calling UDF `buildPostingList`. Though this index creation seems simple in the first place, it usually does not work out of the box. The user also needs to define how to partition large input data sets over the different DBMS nodes. Furthermore, setting up a PDBMS is non-trivial and requires skilled DBAs as also observed in [16][8].

In terms of query processing, most shared-nothing systems strive to partition the input data into balanced partitions at data load time. If necessary, indexes are built locally on the data. Building these indexes is possible only because the DBA has *schema and workload knowledge*. Data sets may also be copartitioned to facilitate join processing again exploiting schema knowledge. Additional join indexes may speed up joins in case copartitioning is not possible. At query time queries are simply split into subqueries and distributed to each node to compute a subset of the result on the local nodes. Intermediate results subsets are then sent to one or multiple merge nodes which assemble the complete result set.

A.2 MapReduce

We need to define our `map` function as follows: `map`(key *DID*, value *content*) $\mapsto$ [(*buzzword*$_1$, *DID*), ..., (*buzzword*$_n$, *DID*)]. This means an input document DID will be mapped to a sequence of *intermediate* output tuples where each intermediate tuple contains a buzzword and the original DID. Non-buzzwords are not output. For each distinct buzzword in the document we generate a separate output tuple. We define `reduce` as follows: `reduce`(key *buzzword*, valueset *DIDset*) $\mapsto$ [(*buzzword* $\oplus$ *postinglist*)]. `reduce` is called once for each distinct buzzword in the set of intermediate tuples. The second parameter `DIDset` contains a set of DIDs containing that buzzword. Thus, the `reduce` function simply needs to form a posting list of those DIDs in this case. Note that this is everything one needs to define in order to build the inverted buzzword search index on arbitrarily large input sets. Everything else will be handled by the MapReduce framework.

I terms of task processing, MapReduce operates in three phases. In the first phase (Map Phase), the framework runs a set of *M map tasks* in parallel where each disjoint subset of the input file is assigned to a particular map task. A map task executes a `map`-call on each input "record" and stores the output locally already partitioned into *R* output files. This means that in total $R \times M$ files will be generated in this phase. In the second phase (Shuffle Phase), the output of the map tasks is grouped and redistributed. Grouping is defined using a hash function `sh` defined on the intermediate key. This guarantees that equal keys from different map tasks are assigned to the same reducer task. In the third phase (Reduce Phase), a set of *R* reduce tasks are run in parallel. Each reduce tasks calls `reduce` for each distinct intermediate key in its input and the set of associated intermediate values. Each reduce tasks writes its output to a single file. Thus the output to the MapReduce task will be distributed over *R* files. See [10] for the original proposal. We will discuss the processing strategy of MapReduce in more detail in Section 2.

[8]Another example for this is the PDBMS Teradata: this company always sets up and configures the system themselves at the customer site to get the performance right.

A.3 Hybrid Approaches

HadoopDB pushes the same SQL query as Parallel DBMS (Section A.1) into local DBMSs in the Map Phase. The local DBMSs in turn compute intermediate results to their local SQL query. Each map task then simply outputs the set of tuples [(*buzzword*$_1$, *DID*), ..., (*buzzword*$_n$, *DID*)] reported by each local DBMS. Finally, HadoopDB uses the same reduce function as MapReduce.

A.4 Hadoop++

Hadoop++ operates exactly as MapReduce by passing the same key-value tuples to the map and reduce functions. However, similarly to HadoopDB, Hadoop++ also allows us:

1. to perform index accesses whenever a MapReduce job can exploit the use of indexes, and
2. to co-partition data so as to allow map tasks to compute joins results locally at query time.

The results of our experiments demonstrate that Hadoop++ can have better performance than HadoopDB. However, in contrast to the latter Hadoop++ does not force users to use SQL and DBMSs.

B. FROM RELATIONAL ALGEBRA TO MAPREDUCE AND BACK

The goal of this section is to show that MapReduce and DBMSs have the same expressiveness. We show that any relational algebra expression can be expressed in MapReduce. Vice versa any MapReduce task may be expressed in extended relational algebra. We extend standard relational algebra by a multimap operator mapping an input item to a set of output items. As a consequence, we conclude that both technologies have the same expressiveness.

This is a formal argument and does *not* imply that plans have to be created *physically* like this. First, we show how to map relational algebra operators to MapReduce (§ B.1 to B.5). Then, we show how to map any MapReduce program to relational algebra (§ B.6).

B.1 Mapping Relational Operators to MapReduce

We assume as inputs two relational input data sets T and S containing items that are termed *records*. The schema of T is denoted $\text{sch}(T) = (a_1, .., a_{ct})$, $\text{sch}(S) = (b_1, .., b_{cs})$ respectively where $a_1, .., a_{ct}$ and $b_1, .., b_{cs}$ are attributes of any domain. In case no schema is known for the data, the schema simply consists of a single attribute containing the byte content of the item to process. In the remainder of this paper we assume that input data sets are split

into records according to the above definition. The subset of attributes in sch(T) representing the key is named $k_T \subseteq$sch(T). The remaining attributes sch(T) $\setminus$ k_T representing the value are named v_T, hence sch(T) = $k_T \oplus v_T$. This also holds for S and we use v_S and k_S accordingly. Inputs and outputs to relational operators are assumed to be duplicate-free sequences, i.e. duplicates are removed unless specified otherwise (e.g. `unionall`). `map` is called for each input record. Key and value are passed as separate parameters and a sequence of intermediate (key, value)-pairs is returned:

$$\texttt{map}(\text{key } k, \text{value } v) \mapsto [(ik_1, iv_1), \ldots, (ik_{m(k,v)}, iv_{m(k,v)})].$$

The number of intermediate output records $m(k, v) \geq 0$ may vary for different k and v. Similarly, `reduce` is called for each distinct intermediate key ik. The set of intermediate values ivs having that intermediate key is passed to `reduce`:

$$\texttt{reduce}(\text{key } ik, \text{vset } ivs) \mapsto [ov_1, .., ov_{r(ik,ivs)}]$$

Thus each `reduce` function produces a sequence of *output* values $ov_1, .., ov_{r(ik,ivs)}$ Again the number of output values $r(ik, ivs) \geq 0$ may vary for different inputs. In many applications the output contains a single value only, i.e. $r(ik, ivs) = 1 \; \forall \; ik, ivs$.

B.2 Unary operators

In the following we will show how to express relational algebra operators using MapReduce. We use $\mapsto$ to denote how to map the left-hand side operator to a MapReduce job. The most simple operator is π. It can be expressed in MapReduce as follows:

Projection (π).

$$\pi_{a_{i_1},..,a_{i_n}}(T) \mapsto \begin{cases} \texttt{map}(\text{key } k, \text{value } v) \mapsto [(\texttt{prj}_{a_{i_1},..,a_{i_n}}(k \oplus v), 1)] \\ \texttt{reduce}(\text{key } ik, \text{vset } ivs) \mapsto [(ik)] \end{cases}$$

Here $\oplus$ denotes that two attributes sets are concatenated to a new schema. `prj`() projects a *single* record to attributes $a_{i_1}, .., a_{i_n}$. Thus π is realized in `map` by concatenating the attributes of the key and the value, projecting to the desired attributes, and outputting the resulting records as the intermediate key. As value we output "1". `reduce` then simply outputs the intermediate key. Recall, that `reduce` is only called once for each intermediate key. Thus our definition of reduce removes all duplicates. Note that the **rename** operator ρ may be defined analogously to π.

The selection operator may be expressed as follows.

Selection (σ).

$$\sigma_P(T) \mapsto \begin{cases} \texttt{map}(\text{key } k, \text{value } v) \mapsto \begin{cases} [(k \oplus v, 1)] & \text{if } P(k \oplus v), \\ \text{none} & \text{else.} \end{cases} \\ \texttt{reduce}(\text{key } ik, \text{vset } ivs) \mapsto [(ik)] \end{cases}$$

Here `map` examines each input record and passes it to the selection predicate P. If P holds, $[(k \oplus v, 1)]$ is output. Otherwise nothing is output. `reduce` simply outputs the intermediate key.

Grouping (Γ). We differentiate between grouping and aggregation. Grouping forms groups of records belonging together. For instance, assume an input T with sch(T) = $\{a_1, a_2\}$ and T = $\{(3,2),(2,1),(1,3),(2,2),(3,4),(1,7)\}$. If we group T, we obtain $\Gamma_{a_1}(T)$ = $\{(3,\{2,4\}),(1,\{3,7\}),(2,\{2,1\})\}$. Only an additional aggregation would transform each group in a_2 into a new value. Hence, grouping may be applied without aggregation.

$$\Gamma_{a_1,..,a_n}(T) \quad \mapsto \quad \begin{cases} \texttt{map}(\text{key } k, \text{value } v) \mapsto \\ [(\texttt{prj}_{a_{i_1},..,a_{i_n}}(k \oplus v), \texttt{prj}_{\text{sch}(T)\setminus\{a_{i_1},..,a_{i_n}\}}(k \oplus v))] \\ \texttt{reduce}(\text{key } ik, \text{vset } ivs) \mapsto [(ik \oplus ivs)] \end{cases}$$

This means `map` concatenates attributes of the key and the value and projects them to the grouping attributes. These attributes are used as the intermediate key. All remaining attributes form the intermediate value. `reduce` then outputs a single record for each distinct intermediate key plus the set of values having that key.

Aggregation (γ). Aggregation can be done by applying an aggregation function $\text{agg}([iv_1, \ldots, iv_n]) \mapsto v$ in `reduce`. If those values were formed by a previous grouping operator, we obtain the desired result:

$$\gamma_{\text{agg}}(T) \mapsto \begin{cases} \texttt{map}(\text{key } k, \text{value } v) \mapsto [(k, v)] \\ \texttt{reduce}(\text{key } ik, \text{vset } ivs) \mapsto [(ik \oplus \text{agg}(ivs))] \end{cases}$$

Another alternative is to combine both grouping and aggregation into a single MapReduce task:

$$\gamma_{\text{agg}}\left(\Gamma_{a_{i_1},..,a_{i_n}}(T)\right) \quad \mapsto \quad \begin{cases} \texttt{map}(\text{key } k, \text{value } v) \mapsto \\ [(\texttt{prj}_{a_{i_1},..,a_{i_n}}(k \oplus v), \texttt{prj}_{\text{sch}(T)\setminus\{a_{i_1},..,a_{i_n}\}}(k \oplus v))] \\ \texttt{reduce}(\text{key } ik, \text{vset } ivs) \mapsto [(ik \oplus \text{agg}(ivs))] \end{cases}$$

This means, we simply modify `reduce` to apply agg() to the output valueset. Note that agg() may be any aggregate including trivial ones such as `MIN`, `MAX`, `SUM`, `AVG`, and `DISTINCT`.

B.3 Binary Operators

MapReduce operates on a single input only. This means that a binary operator cannot be modeled by considering two input files. Therefore, in the following we consider the two inputs to be contained in a single file and denote this as $T|S$. When discussing individual operators we denote this as $T|S$, i.e., T is processed before S. We use a function input($k \oplus v$) $\mapsto$\{ $T|S$ \} to determine whether $k \oplus v$ belongs to T or S. Technically, this function may be implemented by attaching some metadata bit signaling its input to each record. Recall that the precondition for union, intersect, and difference is sch(T) = sch(S).

Union ($\cup$). $T \cup S \mapsto \gamma_{\texttt{distinct}}\left(\Gamma_{\text{sch}(T)}(T|S)\right)$.

This means, we express union as a grouping plus a following duplicate removal on intermediate values. This works as both input sets are already contained in the same input file.

Difference ($\setminus$).

$$T \quad \setminus \quad S \qquad \mapsto \qquad \begin{cases} \texttt{map}(\text{key } k, \text{value } v) \mapsto [(k \oplus v, 1)] \\ \texttt{reduce}(\text{key } ik, \text{vset } ivs) \mapsto \\ \begin{cases} [(ik)] & \text{if } |ivs| = 1 \wedge \text{input}(ik) = T, \\ \text{none} & \text{else.} \end{cases} \end{cases}$$

This means, in `map` we consider all attributes to be intermediate keys. `reduce` tests whether the size of the intermediate valueset ivs contains only a single "1" and ik belongs to input T. Only if this holds, we output ik. Otherwise nothing is output.

Intersection ($\cap$). Obviously intersection may be expressed as $T \setminus (T \setminus S)$ resulting in two MapReduce tasks. However, intersection can also be expressed in a single MapReduce job:

$$T \quad \cap \quad S \qquad \mapsto \qquad \begin{cases} \texttt{map}(\text{key } k, \text{value } v) \mapsto [(k \oplus v, 1)] \\ \texttt{reduce}(\text{key } ik, \text{vset } ivs) \mapsto \\ \begin{cases} [(ik)] & \text{if } |ivs| = 2, \\ \text{none} & \text{else.} \end{cases} \end{cases}$$

This mapping is similar to difference, however we only output a record, if ivs contains two "1"s. As both input sets are duplicate free, this may only hold if the record is contained in both input sets.

Cross Product ($\times$). Let $\text{h}_T()$ be a hash-function defined on the key k_T of sch(T). Let $D > 0$ be a constant. Then the cross product is defined as

$$T \quad \times \quad S \qquad \mapsto \qquad \begin{cases} \texttt{map}(\text{key } k, \text{value } v) \mapsto \\ \begin{cases} [(\text{h}_T(k) \bmod D, k \oplus v)] & \text{if input}(k \oplus v) = T, \\ [(0, k \oplus v), .., (D-1, k \oplus v)] & \text{if input}(k \oplus v) = S. \end{cases} \\ \texttt{reduce}(\text{key } ik, \text{vset } ivs) \mapsto \\ \Big[\texttt{crossproduct}(T_{ik}, S) \mid \\ \quad T_{ik} = \{iv \mid iv \in ivs \wedge \text{input}(iv) = T\}, \\ \quad S = \{iv \mid iv \in ivs \wedge \text{input}(iv) = S\}\Big] \end{cases}$$

Here `map` creates a disjoint partitioning on input T by assigning each record from T a number in $0, .., D-1$. The records from S are replicated by outputting D intermediate records covering all intermediate keys from $0, .., D-1$. The purpose of this partitioning is to

allow for D reduce function calls and thus a concurrent execution on different nodes[9]. Inside a `reduce` call the input set *ivs* is split into two subsets T_{ik} and S. On these sets we then compute the cross product using the *local* function `crossproduct`.

Join ($\bowtie$). Joins may be expressed as $\sigma_{PJ}(T \times S)$ resulting in two MapReduce tasks where one is based on a cross product. Obviously this does not scale for large input sets. We therefore show below a more efficient variant.

$$T \quad \bowtie_{PJ(T,S)} \quad S \quad \mapsto \quad \begin{cases} \texttt{map}(\text{key } k, \text{value } v) \mapsto \\ \begin{cases} [(\texttt{prj}_{a_i}(k \oplus v), k \oplus v)] & \text{if input}(k \oplus v) = T, \\ [(\texttt{prj}_{b_j}(k \oplus v), k \oplus v)] & \text{if input}(k \oplus v) = S. \end{cases} \\ \texttt{reduce}(\text{key } ik, \text{vset } ivs) \mapsto \\ \Big[\texttt{crossproduct}(T_{ik}, S_{ik}) \mid \\ \quad T_{ik} = \{iv \mid iv \in ivs \wedge \text{input}(iv) = T\}, \\ \quad S_{ik} = \{iv \mid iv \in ivs \wedge \text{input}(iv) = S\}\Big] \end{cases}$$

This means that `map` re-partitions both inputs T and S into co-partitions T_{ik} and S_{ik} where the join attributes inside a copartition have the same value. It then suffices to call `crossproduct` for these copartitions. Note that this is similar to a standard relational sort-merge join in the following way: it has to perform a nested-loop, i.e. cross product, on the records having the same value for their join attribute. Also note that for those cases where the join attribute is skewed in a way that the input becomes too large to fit into main memory, the call to `crossproduct` may perform a block-based nested-loop join similar to DBMSs.

B.4 Extended Operators

We discuss an additional operator that does not effect the expressiveness of MapReduce but is useful in the following discussion.

Sort.

$$\texttt{sort}_{a_{i_1},\ldots,a_{i_n}}(T) \quad \mapsto \quad \begin{cases} \texttt{map}(\text{key } k, \text{value } v) \mapsto \\ [(\texttt{prj}_{a_{i_1},\ldots,a_{i_n}}(k \oplus v), \texttt{prj}_{\text{sch}(T)\setminus\{a_{i_1},\ldots,a_{i_n}\}}(k \oplus v))] \\ \texttt{reduce}(\text{key } ik, \text{vset } ivs) \mapsto [\{ik\} \times ivs] \end{cases}$$

This mapping rule is somewhat surprising as neither `map` nor `reduce` perform an actual sort operation. The correctness of this mapping rule is guaranteed as the MapReduce and Hadoop frameworks preserve interesting orders [10]. Finally, let us stress that all other operators (e.g. division, and outer-joins) may be composed by the above operators.

B.5 Relational DAGs

So far we have considered single operators and provided rules to map them to MapReduce jobs. However, relational algebra expressions typically consist of multiple operators forming a *Directed Acyclic Graph* (DAG). These DAGs may be mapped to a cascade of MapReduce jobs by applying our rewrite rules recursively. Outputs of subplans are simply considered input files to the next operator. Thus each operator triggers a separate MapReduce job. For instance, $T \bowtie (S \bowtie U)$ is computed by executing a MapReduce job of §4 for S and U and then executing another MapReduce job on the result and T. Obviously these plans are far from optimal and may be improved in several ways. An upcoming paper discusses how to compute multi-way joins [4]. However those joins do not use co-partitioning as Trojan Join. As discussed above this has severe performance penalties. Therefore it would be interesting to extend our Trojan Join to multiway-joins. We will research this idea as part of future work.

[9]Note that h_T is orthogonal to the partitioning function `sh` (§2). The former hashes records to `reduce` functions, the latter hashes `reduce` functions to reduce tasks.

B.6 Mapping MapReduce to Relational Algebra

The main idea of MapReduce is to perform an aggregation based on two user-defined functions. The purpose of the first function (`map`) is to define the items and grouping key, the purpose of the second function (`reduce`) is to define the aggregation function and the output format. One peculiarity here is that both functions may return a *sequence* of records. To express this we require a special operator that however is straightforward to integrate into an existing relational algebra: a multimap operator.

multimap operator. $\texttt{mmap}_\text{f}(T) \mapsto T'$. For each input item $t \in T$ this operator applies a function $f(t)$ generating zero, one, or multiple output items. All output items have the same schema $\text{sch}(T')$. Function f takes as its argument the entire tuple t, however the attributes of t may be passed as different arguments.

Using this operator we are able to define the backmapping rule for any MapReduce job as follows:

MapReduce Given an input set T and two UDFs, `map` and `reduce`, any MapReduce task can be expressed in relational algebra as:

$$\texttt{MR}_{\texttt{map},\texttt{reduce}}(T) \mapsto \texttt{mmap}_{\texttt{reduce}}\Big(\Gamma_{ik}(\texttt{mmap}_{\texttt{map}}(T))\Big).$$

This means, logically any MapReduce job can be expressed as a multimap operator $\texttt{mmap}_{\texttt{map}}$ followed by a grouping on the intermediate key *ik* and a $\texttt{mmap}_{\texttt{map}}$ using `reduce`. Notice that the main difference of γ and $\texttt{mmap}_{\texttt{reduce}}$ is that the former creates exactly one output value for an input group whereas the latter may create 0, 1, or multiple output values for an input group. Thus, in general $\texttt{mmap}_{\texttt{reduce}}$ *may* aggregate the input value set similarly to γ, but it does not have to.

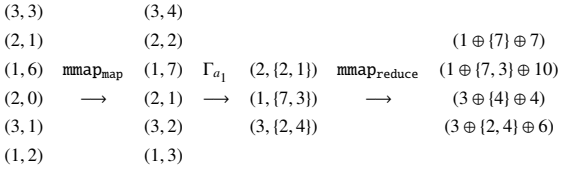


Figure 9: MapReduce processing in relational algebra

Figure 9 shows an example for an input set T having six tuples, $\text{sch}(T) = (a_1, a_2)$ where $k_T = [a_1]$ and $v_T = [a_2]$. The map function increases all values by one, i.e. `map` $:= [k, v + 1]$. The results are then grouped (Γ_{a_1}) and fed into `reduce` which creates all subsets of value sets having a sum greater three, i.e. `reduce` $:= [ik \oplus vset' \oplus \texttt{sum}(vset') \mid vset' \subseteq vset \wedge \texttt{sum}(vset') > 3]$. Each output value is the concatenation of the intermediate key, the subset and its sum. For instance, for tuple $(2, \{2, 1\})$ all subsets have a sum smaller or equal three. Thus no output is produced. For $(1, \{7, 3\})$ two subsets $\{7\}$ and $\{7, 3\}$ have a sum greater than three. Thus two output tuples $(1 \oplus \{7\} \oplus 7)$ and $(1 \oplus \{7, 3\} \oplus 10)$ are produced. Similarly for $(3, \{2, 4\})$ two output tuples are produced.

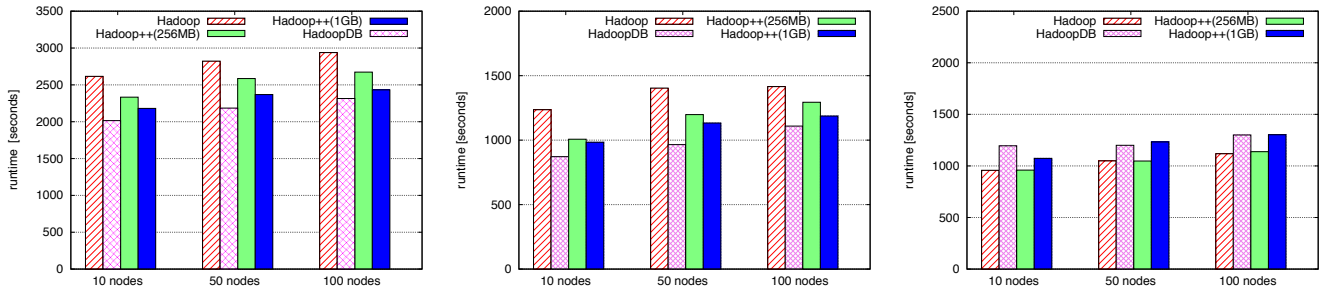
C. SQL QUERIES AND MAPREDUCE JOBS

C.1 Selection Task

SQL query. HadoopDB performs the selection task by executing this SQL statement:

```
SELECT pageURL, pageRank FROM Rankings WHERE pageRank>10;
```

MapReduce jobs. Hadoop performs the same MapReduce job as in [3]. In contrast to Hadoop, Hadoop++ uses a MapReduce job composed of a single `map` function receiving only those $(key, value)$-pairs whose *pageRank* is above 10. This is because Hadoop++ makes use of the SplitTree index.



(a) Large Aggregation Task (b) Small Aggregation Task (c) UDF Aggregation Task

Figure 8: Additional Task Results not related to Indexing and Join Processing

C.2 Join Task

SQL QUERY. HadoopDB pushes the following SQL statement to the local databases. It computes partial aggregates in the local DBMSs and then requires a single `reduce` task for the final aggregation:

```
SELECT sourceIP, COUNT(pageRank),
                 SUM(pageRank), SUM(adRevenue)
FROM Rankings AS R, UserVisits AS UV
WHERE R.pageURL = UV.destURL AND UV.visitDate BETWEEN
Date('2000-01-15') AND Date('2000-01-22')
GROUP BY UV.sourceIP;
```

MAPREDUCE JOBS. While Hadoop uses three MapReduce jobs as explained in [16], Hadoop++ uses a single MapReduce job that implements the co-partitioned join operator explained in §4. This works as follows. First, the selection predicate on *visitDate* is applied and only matching *UserVisits* records are passed to the `map` function. For *Rankings*, all records are passed to the `map` function. The `map` function, in turn, performs the join operation and outputs only those results that satisfy the join predicate. Notice that the `map` function can perform the join operation locally because data in input splits is composed of co-groups from *Rankings* and *UserVisits*. Then, a `combine` performs pre-aggregation before shuffling. Finally, a single `reduce` task performs the final aggregation.

D. SYSTEM SETUP DETAILS

Hadoop. For our experiments, we realized the following changes to the default configuration settings: (1) we stored data into the *Hadoop Distributed File Systems* (HDFS) using 256MB data blocks, (2) we allowed each task tracker to run with a maximum heap size of 1024MB, (3) we increased the sort buffer to 200MB, (4) Hadoop was allowed to reuse the task JVM executor instead of restarting a new process per task, (5) we used 100 concurrent threads for merging intermediate results, (6) we allowed a node to concurrently run two map tasks and a single reduce task, and (7) we set HDFS replication to 1 as done in [3].

Hadoop++ is an improved version of Hadoop that incorporates support for index-scans and co-partitioning as discussed in §3 and §4. We use the same configuration settings as for Hadoop, but we allow it to read data in binary format, except for one of our benchmarks (*UDF task*). A binary record is composed of the data itself and a header containing the lineage of the record and the offset of each attribute.

HadoopDB is a hybrid system combining Hadoop, HBase, and single instance DBMSs, e.g. Postgres, into a system somewhat similar to a PDBMS. We use PostgreSQL 8.4 as local database and increase the memory for shared buffers to 512 MB and the working memory to 1 GB. As in [3], we do not use PostgreSQL's data compression feature, we set data replication to 1 as done in [3].

E. ADDITIONAL BENCHMARK RESULTS

Here we list results for the other tasks defined in the bechmark of [16]. For the grep task, Hadoop++ executes exactly the same code as Hadoop. Therefore runtimes are not effected (and not shown). For the other tasks — even though they are neither related to indexing nor join processing — we still see an improvement of Hadoop++ over Hadoop. We discuss this briefly in the following.

E.1 Large and Small Aggregation Task

These tasks computes the total sum of *adRevenue* grouped by *sourceIP* in *UserVisits*. *Large Aggregation Task* uses all characters of *sourceIP* as grouping key; it computes 2.5 millions groups. In contrast, *Small Aggregation Task* uses the first seven characters of *sourceIP* as grouping key; it computes 2,000 groups.

Figures 8(a) and 8(b) summarize the results for this task. We observe that for both tasks (small and large and aggregation) Hadoop++ outperforms Hadoop because it reads data in binary format and, hence, it can read *sourceIP* and *adRevenue* without reading other attributes. Furthermore, we can observe that Hadoop++ is slower than HadoopDB, because Postgres applies a hash aggregation while Hadoop++ uses a sort-based-aggregation. However, the performance of Hadoop++ and HadoopDB is in the same ballpark even though HadoopDB emulates a non-compressed PDBMS and even though the improvements of Hadoop++ are not related to aggregate computation.

E.2 UDF Aggregation Task

We also consider an aggregation query that parses each HTML document in *Documents*, using a UDF[10], which extracts the inlinks and counts the number of unique pages referencing a URL.

When loading HTML documents into HDFS for Hadoop and Hadoop++, we proceed as in [3, 16], i.e. we concatenate several documents into larger ones in order to avoid memory problems with the HDFS' server when dealing with a large number of documents. In contrast, HadoopDB stores each HTML document separately in relation *Documents*.

Figure 8(c) shows results for this task. We see the best variant of Hadoop++ is at least as good or better than HadoopDB. This is because Hadoop++ processes concatenated HTML documents as described above. Compared to Hadoop, Hadoop++(256MB) has similar performance. However, overall, neither HadoopDB nor Hadoop++ can improve over the Hadoop for this particular task.

[10]Not to be confused with the ten UDFs provided by Hadoop as explained above.