UNIVERSITY OF CALIFORNIA,
IRVINE

Efficient Processing of Set-Similarity Joins on Large Clusters

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Rares Vernica

Dissertation Committee:
Professor Chen Li, Co-Chair
Professor Michael J. Carey, Co-Chair
Professor Sharad Mehrotra

2011

DEDICATION

To my parents, to whom I owe everything I am.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Page

ACKNOWLEDGMENTS

I owe my deepest gratitude to my advisers Professor Chen Li and Professor Michael J. Carey.

Professor Chen Li took the challenge of shaping me into an independent researcher at the beginning of my Ph.D. studies. He taught me how to read and write research studies. He showed me how to formulate, motivate, solve, and present research problems. He taught me how to give great presentations and how to turn my research into useful products.

Professor Michael J. Carey joined Professor Chen Li in shaping me into an independent researcher. He provided a great deal of inspiration and motivation. His guidance in building systems was invaluable. He taught me how to analyze results and how to find the real reasons behind a system's behavior.

I am very grateful to my mentors and colleagues from development and research labs who took the challenge of broadening my knowledge with industry experience. I owe special thanks to my mentor Andrey Balmin, my manager Eugene J. Shekita and my colleagues at IBM. Their great mentorship and guidance made Chapter 4 possible. Partha Saha and Kazi Zaman mentored and guided me during my internship at Yahoo! Venkatesh Ganti mentored me during my internship at Microsoft Research.

I would also like to thank my co-authors and colleagues for helpful feedback and discussions. In particular I would like to thank Alexander Behm, Vinayak R. Borkar, Yingyi Bu, Shengyue Ji, and Nicola Onose.

This research is partially supported by the National Science Foundation IIS awards 0238586, 0331707, 0844574, 0910989, and 1030002, by the University of California, Irvine, Department of Computer Science Chair's Fellowship, by the California Institute for Telecommunications and Information Technology Fellowship, and by grants from eBay, Google, IBM, Microsoft and Yahoo!

Part of the text of this dissertation is a reprint of the material as it appears in Efficient parallel set-similarity joins using MapReduce, Rares Vernica, Michael J. Carey, Chen Li, SIGMOD Conference 2010: 495-506. The co-authors listed in this publication directed and supervised research which forms the basis for the dissertation.

CURRICULUM VITAE

Rares Vernica

EDUCATION

Doctor of Philosophy in Information and Computer Science — **2011**
University of California, Irvine — *Irvine, California*

Bachelor of Science in Computer Science and Engineering — **2004**
Politehnica University of Bucharest — *Bucharest, Romania*

SELECTED HONORS AND AWARDS

Key Scientific Challenges Award — **2010**
Yahoo! Inc.

Chair's Fellowship — **2005–2006**
Department of Computer Science
University of California, Irvine

Calit^2 Fellowship — **2005**
California Institute for Telecommunications and Information Technology

PUBLICATIONS

CIRCUMFLEX: A Scheduling Optimizer for MapReduce Workloads With Shared Scans **2011**
Workshop on Large Scale Distributed Systems and Middleware (LADIS)
International Conference on Very Large Data Bases (VLDB)

ASTERIX: Towards a Scalable, Semistructured Data Platform for Evolving World Models **2011**
Journal of Distributed and Parallel Databases

Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing **2011**
IEEE International Conference on Data Engineering (ICDE)

AKYRA: Efficient Keyword-Query Cleaning in Relational Databases **2010**
Technical Report, University of California, Irvine

Efficient Parallel Set-Similarity Joins Using MapReduce **2010**
ACM Special Interest Group on Management of Data (SIGMOD)

Efficient Top-k Algorithms for Fuzzy Search in String Collections **2009**
Workshop on Keyword Search on Structured Data (KEYS)
ACM Special Interest Group on Management of Data (SIGMOD)

Entity Categorization Over Large Document Collections **2008**
ACM Special Interest Group on Knowledge Discovery
and Data Mining (KDD)

SEPIA: Estimating Selectivities of Approximate String Predicates in Large Databases **2008**
International Journal on Very Large Data Bases (VLDB J.)

Relaxing Join and Selection Queries **2006**
International Conference on Very Large Data Bases (VLDB)

ABSTRACT OF THE DISSERTATION

Efficient Processing of Set-Similarity Joins on Large Clusters

By

Rares Vernica

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 2011

Professor Chen Li, Co-Chair
Professor Michael J. Carey, Co-Chair

Joining two datasets where the join condition is based on set similarity instead of equality is very important in many applications, such as social networks, master data management, plagiarism detection, and query refinement based on users' similar preferences. For instance, finding Web site members with similar interests and removing duplicate customers after company acquisitions are two examples. Additionally, many applications need to deal with a large amount of data that cannot be processed by a single machine. Using clusters of machines where algorithms are executed in parallel is a more scalable approach. Recently, scalable, shared-nothing, data-intensive computation platforms have gained attention from both industry and academia. This trend started with the MapReduce framework, and more general frameworks have been proposed recently.

In this thesis we study how to efficiently perform set-similarity joins, a.k.a. fuzzy-joins, on large shared-nothing clusters. We propose a three-stage approach for computing end-to-end set-similarity joins. The input to the approach is a set of records (or two sets of records), and the output is a set of joined records based on the desired set-similarity condition. The proposed approach efficiently partitions fuzzy-join pro-

cessing across many nodes, balancing the workload and minimizing the need for data replication.

This thesis makes three contributions related to the proposed three-stage approach to compute fuzzy-joins. First, we study fuzzy-joins in the context of the popular MapReduce framework. Next, we propose a set of improvements to the MapReduce runtime model to improve its handling of fuzzy-join queries as well as other types of queries. Finally, we study fuzzy-joins in the context of a new data-intensive computing platform being developed at UC Irvine, ASTERIX. We show how to integrate fuzzy-joins into the system starting at the query language level and ending in efficient execution. Along with each of the contributions, we provide performance results from experiments on real datasets, synthetically increased in size, to study the speedup and scaleup properties of our techniques.

Chapter 1

Introduction

There are many applications that require detecting similar pairs of records where the records contain string or set-based data. A list of possible applications includes: detecting near duplicate web-pages in web crawling [35], document clustering [15], plagiarism detection [36], master data management, making recommendations to users based on their similarity to other users in query refinement [55], mining in social networking sites [58], and identifying coalitions of click fraudsters in online advertising [48]. For example, in master-data-management applications, a system has to identify that names "`John W. Smith`", "`Smith, John`", and "`John William Smith`" are potentially referring to the same person. As another example, when mining social networking sites where users' preferences are stored as bit vectors (where a "`1`" bit means interest in a certain domain), applications want to use the fact that a user with a preference bit vector "`[1,0,0,1,1,0,1,0,0,1]`" possibly has similar interests to a user with preferences "`[1,0,0,0,1,0,1,0,1,1]`", since the two bit vectors are similar. Specifically websites like `www.yellowpages.com` (a business listing website) need to identify near-duplicate business listings based on business name and address, websites like `www.ask.com` (a web search engine) need to identify near-duplicate web

pages based on their text content, and websites like `www.eharmony.com` (an online dating website) need to identify similar users based on their profiles.

Detecting such similar pairs is challenging today, as there is an increasing trend of applications being expected to deal with vast amounts of data that usually do not fit on one machine. For example, the U.S. Department of Homeland Security database has 100 million person identities [34]; the GeneBank dataset [30] contains 100 billion bases and 100 million sequences; the Google N-gram dataset [60] has 1 trillion records. Applications with such datasets usually make use of clusters of machines and employ parallel algorithms in order to efficiently deal with this vast amount of data.

When dealing with a very large amount of data, detecting similar pairs of records becomes a challenging problem, even if a large computational cluster is available. Parallel data-processing paradigms rely on data partitioning and redistribution for efficient query execution. Partitioning records to find similar pairs of records is challenging for string or set-based data, as hash-based partitioning using the entire string or set does not suffice.

In this thesis we study how to efficiently perform set-similarity matching on large computer clusters. The core of our study is based on the development of a three-stage approach for computing end-to-end set-similarity joins. We also call this kind of joins fuzzy-joins. The input to the approach is a set of records (or two sets of records), and the output is a set of joined records based on a set-similarity condition. The proposed approach efficiently partitions the processing across many nodes, balancing the workload and minimizing the need for data replication. In this thesis we focus exclusively on set-similarity joins, which are part of a more general class of joins called fuzzy-joins. We use set-similarity join and fuzzy-join interchangeably in this thesis.

We study such fuzzy-joins in the context of two systems. First, we study the problem

in the context of the popular MapReduce framework. For each of the three stages in our approach we propose two MapReduce implementations. We start with the self-join case, finding similar pairs in a single dataset, and then generalize our techniques for the R-S join case, finding similar pairs from two datasets. We also present techniques for dealing with insufficient main memory. The speedup and scale-up characteristics of each of the implementations is studied. Then, in order to improve the performance of the MapReduce implementation, we propose a set of techniques to make query execution in MapReduce more adaptive to the data. We present a set of techniques that integrate seamlessly with the MapReduce framework and provide significant benefits to fuzzy-join queries. These techniques are general and applicable to other types of queries as well. We present a performance study to show their benefits.

Second, we study fuzzy-joins in the context of a new data-intensive computing platform being developed at UC Irvine, UC San Diego, and UC Riverside, called ASTERIX [11]. We first study fuzzy-join processing using the ASTERIX's runtime platform, Hyracks. We propose a Hyracks-native implementation of our fuzzy-join techniques. We analyze the relative performance of the MapReduce implementation and the native implementation of fuzzy-joins. Next, we integrate fuzzy-joins into the high-level query language of ASTERIX, AQL. We present a set of query-processing techniques to support fuzzy-joins in AQL with performance and generality as their main goals. We experimentally compare the Hyracks native approach with the Hyracks plan generated from AQL.

This thesis is organized as follows: In Chapter 2 we present the foundations of set-similarity joins and shared-nothing parallel data processing. We also describe our overall approach to parallel fuzzy-join evaluation. Next, in Chapter 3 we present our treatment of fuzzy-joins in the MapReduce context. In Chapter 4 we propose our set

of adaptive techniques for the MapReduce framework that improve the processing of fuzzy-join queries and also varying other common queries. Next, in Chapter 5 we describe the integration of fuzzy-join query processing with a more general runtime than MapReduce, namely Hyracks, and with a high-level query language, AQL. We present future work and conclude this thesis in Chapter 6.

Chapter 2

Foundations

In this thesis we focus on the problem of identifying similar records based on a set-similarity function.

Problem statement: Given two files of records, R and S, a set-similarity function, sim, and a similarity threshold τ, we define the set-similarity join of R and S on columns $R.a$ and $S.a$ as finding and combining all pairs of records (r, s) from R and S where $sim(r.a, s.a) \geq \tau$.

One example application for set-similarity joins is identifying similar records based on string similarity. For string similarity, we first convert strings into sets by tokenizing them. Examples of tokens are words or q-grams (overlapping sub-strings of a fixed length). Since some tokens may occur multiple times in a string, we treat each occurrence of the same token as a new token by appending a counter to each token. In this way, each string is converted to a set of tokens. For example, the string "`I will call you back, I promise`" can be tokenized into the word set {`I_1, will_1, call_1, you_1, back_1, I_2, promise_1`}. In order to measure the similarity between strings, we use a set-similarity function such as the Jaccard

or Tanimoto coefficient, cosine coefficient, etc[1]. For example, the Jaccard similarity function for two sets x and y is defined as: $jaccard(x, y) = \frac{|x \cap y|}{|x \cup y|}$. Thus, the Jaccard similarity between the strings "`I will call you back, I promise`" and "`I will call you soon`" is $\frac{4}{8} = 0.5$.

Another example application for set-similarity joins is identifying similar records based on the similarity of a set-valued column, such as the social networking application described in Chapter 1 where users' preferences are stored as bit vectors. In this type of applications, tokenization is not necessary and a set-similarity function can be applied directly on the input sets.

In this chapter we first present the foundations of set-similarity join query processing and data-intensive computing platforms. In the second part of the chapter, we present the core of our approach to parallel set-similarity join processing.

2.1 Set-Similarity Joins

Set-similarity joins on a single machine have been widely studied in the literature recently [6, 10, 18, 56, 64]. Inverted-list-based algorithms for finding pairs of strings that share a certain number of tokens in common have been proposed in [56]. After such pairs are identified, they have to be verified using the desired set-similarity function. Later work has proposed various filters that can help decrease the number of pairs that need to be verified. A technique called prefix filter was proposed in [18]. This filter will be further discussed in Section 2.1.1. The length filter has been studied in [6, 10]. The main idea of this filter is that similar sets will have similar lengths. That is, given the length of a set we can compute a length range within which similar

[1]The techniques described in this thesis can also be used for approximate string search using the edit or Levenshtein distance [33].

sets have to be. Instead of directly using the tokens, the approach in [6] generates a set of signatures based on the tokens and relies on the fact that similar strings need to have a common signature. Two other filters, namely positional filter and suffix filter, were proposed in [64]. The positional filter extends the prefix filter by taking into consideration the positions of the matching tokens in the prefix. That is, the relative distance of the matching tokens, ordered based on a global order, needs to be within a certain threshold. The suffix filter is a generalization of the positional filter for the set suffixes. For edit distance in particular, two more filters based on mismatching tokens between two sets have been proposed in [63]. These two filters look at the location and the content of the mismatching tokens and return an edit-distance lower bound between the two sets. Different ways of formulating the similarity problem have been proposed in [31, 62]. The algorithms in [62] are designed only to return the top-k pairs of records ranked by their similarities. [31] studied the problem of returning good partial answers by using the idea of locality-sensitive hashing [38]. It is worth noting that most of the existing algorithms have dealt with values that have already been projected on their similarity attribute and the algorithms produce only the list of similar RIDs. In this thesis, we provide algorithms for answering set-similarity self-join queries *end-to-end*, where we start from records containing more than just the join attribute and end with actual pairs of joined records.

2.1.1 Prefix Filtering

As discussed, efficient set-similarity join algorithms rely on effective filters to decrease the number of candidate pairs whose similarity needs to be verified. In the past few years, there have been several studies involving *prefix filtering* [10, 18, 64]; prefix filtering is based on the pigeonhole principle and works as follows. The set elements are first ordered based on some global token order. For each set, its *prefix* of length

n is then defined to be the first n tokens of its ordered set of tokens. The prefix filtering principle states that similar strings must share at least one common token *in their prefixes.* The length of the prefix required for this principle to hold depends on the size of the token sets, the similarity function, and the similarity threshold. For example, given the string s, "`I will call you soon`", and the global token order {`soon, call, will, you, I`}, the prefix of length 2 of s is `[soon, call]`. Using this principle, a set-similarity join between two relations with a set-valued attribute may be computed as follows. Records of one relation are indexed on the individual values of the tokens in their prefixes. Then, using the prefix tokens of the records in the second relation, we can probe the first relation and generate candidate pairs. The prefix filtering principle gives only a necessary condition for similar records, so the generated candidate pairs need to be verified as a final check. Good performance can be achieved when the global token order corresponds to an increasing token-frequency order, since fewer candidate pairs are likely to be generated in this case.

A recent state-of-the-art algorithm in the set-similarity join literature is the PPJoin+ technique presented in [64]. It uses the prefix filter along with the length filter, the positional filter, and the suffix filter. The PPJoin+ technique provides a good solution for answering set-similarity join queries on one node, and we will utilize it here a potential building block for a parallel solution.

2.2 Data-Intensive Computing Platforms

Recently, data-intensive computing platforms have been getting a lot of attention in both the industry [3, 12, 21, 23, 29, 59] and the academia [9, 11, 14, 16, 20, 28, 39, 43, 49, 69, 71]. Currently, two powerful trends dominate the space. The first trend is the MapReduce trend. This trend is centered around the MapReduce framework

introduced by Google [24]. The MapReduce framework has gained significant popularity with the help of its popular open-source implementation called Hadoop [3]. In Hadoop, parallel computations are expressed by simply implementing a `map` function and a `reduce` function in Java. Problems are then solved in a handful of high-level query languages that have been proposed for this framework [12, 51, 59]. The second trend in this space is centered around frameworks that provide a greater flexibility in the way that computations are expressed. Such frameworks usually provide, as building blocks, more than just a `map` and `reduce` interface [9, 14, 39, 69]. They too usually offer one or more associated high-level query languages [11, 67, 71].

In this section we will give a brief introduction to both trends. For the non-MapReduce trend, as an example, we will briefly describe the ASTERIX project. Other non-MapReduce projects follow similar patterns.

2.2.1 MapReduce

The MapReduce parallel data-processing framework [22], pioneered by Google, is quickly gaining popularity in the industry [3, 4, 12, 21, 23, 40, 29, 50, 59] and the academia [1, 7, 16, 20, 28, 43, 47, 49, 53, 66, 70]. Hadoop is the dominant open-source MapReduce implementation and is backed by Yahoo!, Facebook, and others. Example applications for the MapReduce paradigm include processing and indexing crawled documents, analyzing Web request logs, etc.

In MapReduce, data is initially partitioned across the nodes of a cluster and stored in a distributed file system (DFS). Data content is represented as `(key, value)` pairs. The computation is then expressed using two functions:

```
map    (k1,v1)         → list(k2,v2);
reduce (k2,list(v2)) → list(k3,v3).
```

The `map` function takes as input a `(key, value)` pair, denoted by `(k1, v1)`, and produces as output a list of new `(key, value)` pairs, denoted by `list(k2, v2)`. The `reduce` function takes as the input one of the keys output from the map (`k2`) and a list containing all the values output with that key (`list(v2)`). In return, the reduce function outputs a new list of `(key, value)` pairs, denoted by `list(k3, v3)`.

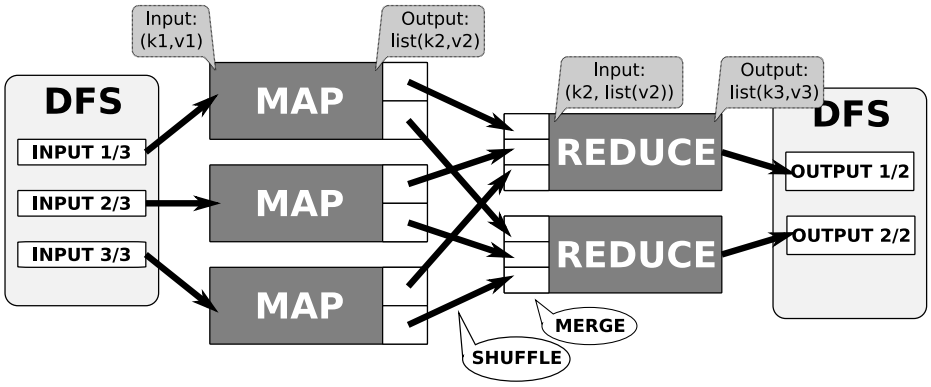


Figure 2.1: Data flow in a MapReduce computation.

Figure 2.1 shows the data flow in a parallel MapReduce computation. The computation starts with a map phase in which the `map` function is applied in parallel on different partitions of the input data, called "splits". A map task, or mapper, is started for every split, and it iterates over its input `(key, value)` pairs applying the map function. The `(key, value)` pairs output by each mapper are then sorted and hash-partitioned on the `key`. In Hadoop, the pairs are sorted by their partition number and the `key` in a fixed-size memory buffer. Once the buffer is filled up, the sorted run, called a *spill* is written to the local disk. At the end of the mapper execution all the spills are merged into a single sorted file. At each receiving node, a reduce task, or reducer, fetches all of its sorted partitions during the shuffle phase and merges them into a single sorted stream. For each key, a reduce call is made and all the pair

values that share that key are streamed-by. The output of each `reduce` function is written to a distributed file in on the Distributed File System (DFS).

Besides the `map` and `reduce` functions, the MapReduce framework also allows the user to provide a `combine` function that is executed on the same nodes as the mappers right after the `map` functions have finished. This function acts as a local reducer, operating on the local `(key, value)` pairs. This function allows the user to decrease the amount of data to be sent through the network. The signature of the `combine` function is:

$$\texttt{combine (k2,list(v2))} \rightarrow \texttt{list(k2,v2).}$$

In Hadoop, the `combine` function is applied once the outputs have been sorted in the memory, just before they are spilled to disk. At the end of the map execution, when all the spills are merged into a single output file, the combiner function is applied again on the merged results. Figure 2.2 shows the sequence of operations necessary for applying combiners.

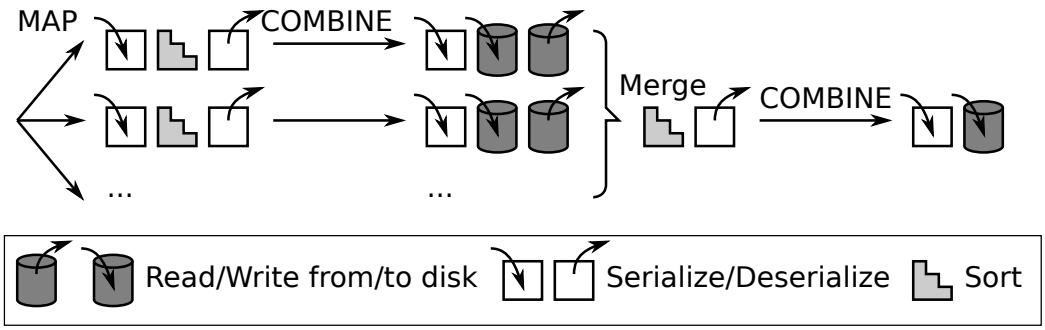


Figure 2.2: Serialize, deserialize, disk read, disk write and sort operations performed to apply the `combine` function.

Each node in a Hadoop cluster has a fixed number of "slots" for executing map and reduce tasks. A node will never have more map or reduce tasks running concurrently

than the corresponding number of slots. If the number of mappers of a job exceeds the number of available map slots, the job runs with multiple "waves" of mappers. Similarly, the reducers may also run in multiple waves.

Finally, the MapReduce framework also allows the user to provide initialization and tear-down functions for each MapReduce function and allows them to customize the hashing and comparison functions to be used when partitioning and sorting the keys.

A number of techniques have been proposed to improve the performance of MapReduce jobs [20, 28, 43, 46, 49]. In [43], the authors propose a set of general low-level optimizations which include improving I/O speed for local data, exploiting indexes, using different decoding schemes when deserializing the data, using fingerprinting for faster key comparisons, and block-size tuning. The authors of [28] observed that collocating data blocks in the distributed file system and adding an index to each block substantially improves the performance of join algorithms. The study in [49] focused on grouping MapReduce jobs that perform common computations and evaluating each group as a single job. In [20], the authors modified the MapReduce architecture to allow pipelining of the intermediate data between operators. More recently, in [46], the authors studied the problem of using MapReduce for one-pass analytics.

A number of higher-level languages have been proposed on top of MapReduce, including Hive [59], which is an SQL-like language, Jaql [12], which is a Java-like language, and Pig [29], which is a relational-algebra-like language. All these languages could potentially benefit from the addition of a set-similarity join operator based on our techniques. In fact, regarding set-similarity joins in MapReduce, which are presented in detail in Chapter 3, in the context of Jaql [12], a Jaql tutorial based on fuzzy joins was presented in [42].

2.2.2 ASTERIX

The ASTERIX project [11] is an ongoing, multi-UC effort to design, develop, and deliver in open source a highly scalable platform for information storage and data-intensive information analysis. Essentially, the ASTERIX platform aims to be a parallel semistructured data management system that is able to ingest, store, index, query, analyze, and publish massive quantities of semistructured information.

ASTERIX Data Model and Query Language

To support a wide variety of semistructured data formats, ASTERIX data storage and query processing are based on a semistructured data model called the ASTERIX Data Model (ADM). Each individual ADM data instance is typed and can optionally by self-describing. All data instances live in datasets (the ASTERIX analogy to tables), and datasets can be indexed, partitioned, and possibly replicated to achieve scalability and availability. External datasets (i.e., datasets that reside in files that are not under ASTERIX control) are also supported. Datasets may have associated schema information describing the core content of their instances. ASTERIX schemas are by default "open," in the sense that individual data instances may contain more information than what their dataset schema indicates and can differ from one another regarding their extended content. Data is accessed and manipulated through the use of the associated ASTERIX Query Language (AQL). AQL is designed to cleanly match and handle the data structuring constructs of ADM. It is inspired by XQuery [65], but it omits the many XML-specific and document-specific features of XQuery. AQL and ADM will be covered in more details in Chapter 5

ASTERIX System Architecture and Runtime Logic

The ASTERIX data storage, query processing, and computational capabilities are targeted at large, shared-nothing, commodity clusters. Figure 2.3 presents an overview of the main software components of ASTERIX. Its software architecture contains most of the "usual suspects," including a separation of query compilation from runtime plan execution. Areas where ASTERIX is different than other platforms include the compilation/execution division of labor as well as the management of data. Regarding the division of labor, as will be discussed more in Chapter 5, AQL requests are compiled into jobs for an ASTERIX execution layer called Hyracks. ASTERIX concerns itself with the data details of AQL and ADM, turning AQL requests into Hyracks jobs, while Hyracks determines and oversees the utilization of parallelism based on information and constraints associated with the resulting jobs' operators as well as on the runtime state of the cluster. Hyracks itself is intended to support AQL plans, MapReduce-style computation, and more general operator graphs in between, and is therefore multi-purpose in terms of its target customers or end users. Regarding data management, the majority of today's data-intensive computing platforms focus on the analysis of data that is "just passing through." In contrast, more like a parallel DBMS, ASTERIX stores and manages large volumes of data and aims to simultaneously support short queries, longer analyses, and ongoing updates (the arrival of new versions) to the data in its datasets.

Hyracks Overview

Hyracks is a generalized alternative to infrastructures such as MapReduce, Hadoop, and Dryad for solving data-parallel problems. Hyracks' level of expressiveness goes beyond MapReduce, which offers a very limited programming model based on a few

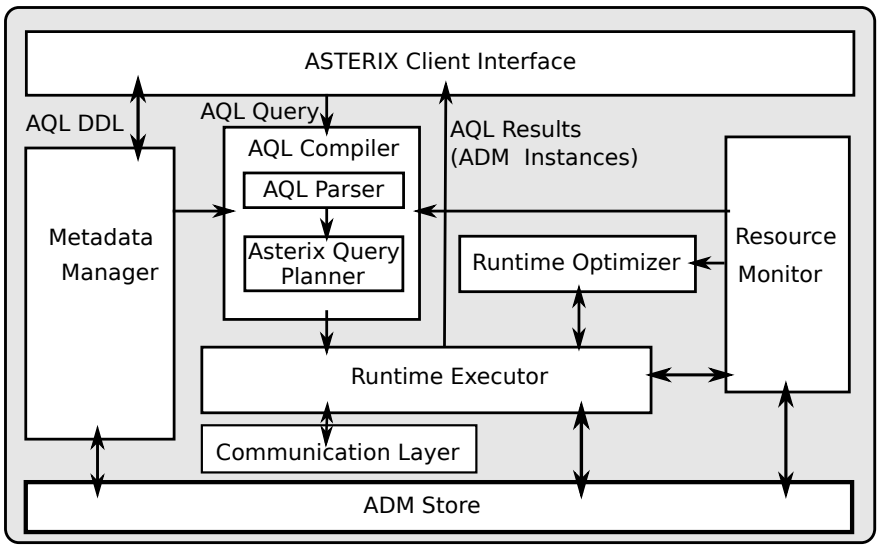


Figure 2.3: ASTERIX system architecture

user-provided functions, while providing out-of-the-box support for many commonly occurring communication patterns and operators needed in data-oriented tasks.

Hyracks has been designed to run on a cluster of commodity computers. To do so in a robust manner, it has built-in support to detect and recover from possible system failures that might occur during the evaluation of a job through the use of heartbeats. A Hyracks cluster has two kinds of nodes: *cluster controllers* and *node controllers*. The cluster controllers monitor the health of the node controllers while the latter are used to evaluate user jobs. Cluster controllers monitor system's resource usage and schedule the execution of pieces of Hyracks jobs. Cluster controllers are also responsible for interacting with clients and providing them with a single system image (SSI). Hyracks will be covered in more detail in Chapter 5.

2.3 Parallel Set-Similarity Join

We now turn our attention to the problem of scaling up set-similarity join processing. One of the main issues when computing set-similarity joins in a parallel setting is to decide how data should be partitioned and replicated. The main idea of our algorithms is the following. Parallel equi-join algorithms partition the data across the network by hashing on the join value [25, 26, 45]. In our case, as we are interested in pairs of records with similar join-attribute values, where the value is actually a set, we cannot directly hash-partition on the value. Instead, we hash-partition on (possibly multiple) *tokens* generated from the value. Tokens are defined such that similar attribute values have at least one token in common. Possible example tokens include: the elements of a set and the word tokens of a string. For instance, the string "`I will call you soon`" would have four word-based tokens: "`I`", "`will`", "`call`", "`your`" and "`soon`". To leverage the prefix filtering principle, we use a global order on the set of tokens and only hash-partition on the prefix tokens. Moreover, we will choose the token order to be the ascending token-frequency order, as suggested by the prefix filtering principle [18].

We divide the parallel processing of fuzzy-join queries into three stages:

- **Stage 1:** Compute the global token order. Tokens are ordered by increasing frequency, so that the least popular tokens are first. This stage computes the frequency of each token and orders the tokens accordingly.
- **Stage 2:** Extract the record IDs ("RIDs") and the join-attribute values from each record, tokenize the join-attribute value, and re-distribute the RID and join-attribute value pairs based on the prefix tokens. After the data redistribution, each machine computes the similarity of its local join-attribute values and outputs RID pairs of the similar records.

- **Stage 3:** Generate actual pairs of joined records. This stage uses the list of RID pairs from the second stage with the original data to build the final pairs of similar records.

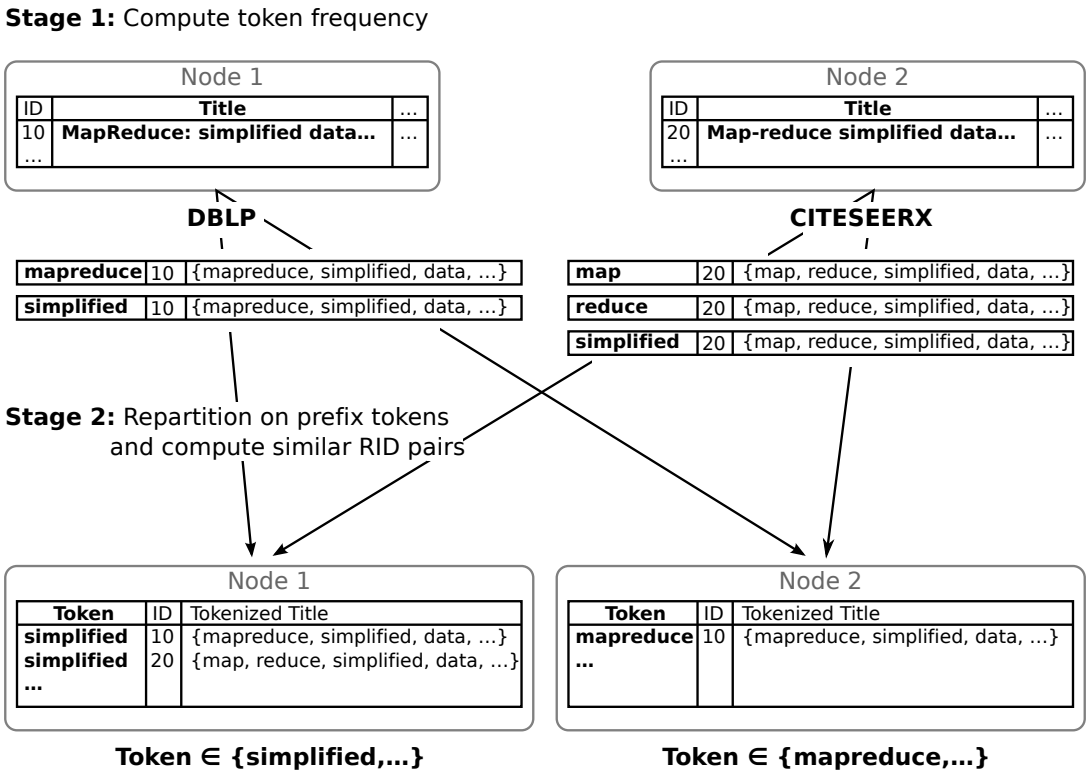


Figure 2.4: Parallel set-similarity join processing stages for joining two publications datasets, "DBLP" and "CITESEERX", on publication title

Figure 2.4 summarizes these stages and shows the intuition behind the data repartitioning strategy in Stage 2 for a set-similarity join between two publications datasets, "DBLP" and "CITESEERX". For both datasets, each record contains the record ID, the publication title, and other information. The set-similarity join is on publication title. That is, we are looking of for records with similar sets of words in the title. Figure 2.5 shows example inputs and outputs for each of the three stages for the same query.

In this thesis we bring together the two worlds of data-intensive parallel computing and fuzzy-joins while focusing on performance, scalability, and generality. We provide

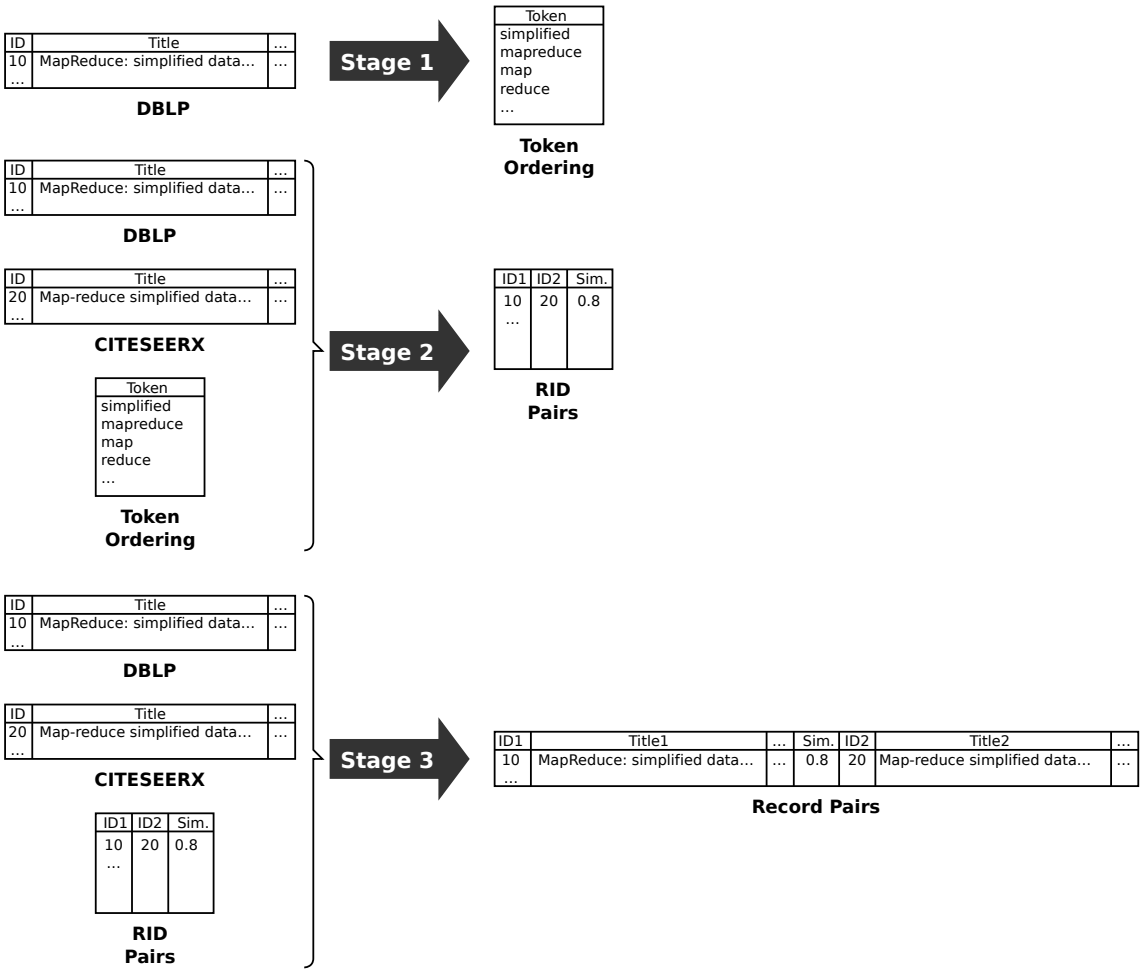


Figure 2.5: Example inputs and outputs for each of the three parallel-set-similarity-join-processing stages for joining two publications datasets, "DBLP" and "CITESEERX", on publication title

complete, general, and parallel end-to-end solutions for set-similarity joins on large clusters.

Chapter 3

Efficient Set-Similarity Joins in MapReduce

3.1 Introduction

In this chapter, we focus on finding similar pairs of records using the MapReduce framework as the parallel data-processing paradigm. The contributions of this chapter are as follows:

- We describe efficient solutions for answering set-similarity queries that exploit the MapReduce framework. We show how to efficiently deal with problems such as partitioning, replication, and multiple inputs by manipulating the keys used to route the data in the framework.
- We present MapReduce-specific methods for controlling the amount of data kept in memory during a join by exploiting the properties of the data that needs to be joined.

- We provide algorithms for answering set-similarity self-join queries *end-to-end*, where we start from records containing more than just the join attribute and end with actual pairs of joined records (not just their RIDs).

- We present set-similarity self-join algorithms and show how they can be extended to answer set-similarity join queries between two different relations.

- We present MapReduce-specific strategies for handling the exceptional situation where, even if we use the finest-granularity partitioning method, the data for each node is too large to fit into the node's main memory.

The rest of the chapter is structured as follows. In Section 3.2 we present a family of MapReduce set-similarity join algorithms for the self-join case, while in Section 3.3 we show how the algorithms can be extended to the R-S join case. Next, in Section 3.4, we present strategies for handling the insufficient-memory case in MapReduce. A performance evaluation is presented in Section 3.5 and we conclude in Section 3.6.

3.2 Self-Join Case

In this section we present MapReduce techniques for the set-similarity self-join case. As outlined in the previous chapter, the solution is divided into three stages. In the following, we present our MapReduce implementation for each of the three stages: Token Ordering, RID-Pair Generation, and Record Join. We take as input a set of records stored in the distributed file system, a join column, a similarity function, and a similarity threshold. We output a set of record pairs that pass the similarity threshold on the join column.

3.2.1 Stage 1: Token Ordering

We consider two possible alternate methods for ordering the tokens in the first stage. Both methods take as input the original records and produce a list of the tokens that appear in their join-attributes ordered increasingly by their global frequency of usage.

Basic Token Ordering (BTO)

Our first approach, called Basic Token Ordering ("BTO"), relies on two MapReduce phases. The first phase computes the frequency of each token and the second phase sorts the tokens based on their frequencies. Algorithm 1 shows the pseudo-code for this approach. In the first phase, the `map` function gets as input the original records. For each record, the function extracts the value of the join attribute and tokenizes it. Each token produces a `(token, 1)` pair. To minimize the network traffic between the `map` and `reduce` functions, we use a `combine` function to aggregate the 1's output by the `map` function into partial counts. Figure 3.1 shows the data flow for an example dataset, self-joined on an attribute called "`a`". In the figure, for the record with RID 1, the join-attribute value is "`A B C`", which is tokenized as "`A`", "`B`", and "`C`". Subsequently, the `reduce` function computes the total count for each token and outputs `(token, count)` pairs, where "`count`" is the total frequency for the token.

The second phase uses MapReduce to sort the pairs of tokens and frequencies from the first phase. The `map` function swaps the input keys and values so that the input pairs of the `reduce` function are sorted based on their frequencies. This phase uses exactly one reducer so that the result is a totally ordered list of tokens.

Algorithm 1: Basic Token Ordering (BTO)

```
   // --- - Phase 1 - ---
 1 map (k1=unused, v1=record)
 2 |  extract join attribute from record;
 3 |  foreach token in join attribute do
 4 |  |  output (k2=token, v2=1);
   |
 5 combine (k2=token, list(v2)=list(1))
 6 |  partial_count ← sum of 1s;
 7 |  output (k2=token, v2=partial_count);
   |
 8 reduce (k2=token, list(v2)=list(partial_count))
 9 |  total_count ← sum of partial_counts;
10 |  output (k3=token, v3=total_count);
   |
   // --- - Phase 2 - ---
11 map (k1=token, v1=total_count)
   |  // swap token with total_count
12 |  output (k2=total_count, v3=token);
   |
   /* only one reduce task; with the total count as the key and only
      one reduce task, tokens will end up being totally sorted by
      token count                                                    */
13 reduce (k2=total_count, list(v2)=list(token))
14 |  foreach token do output (k3=token, v3=null);
```

One-Phase Token Ordering (OPTO)

An alternative approach to token ordering is to use just one MapReduce phase. This approach, called One-Phase Token Ordering ("OPTO"), exploits the fact that the list of tokens could be much smaller than the original data size. Instead of using MapReduce to sort the tokens, we can explicitly sort the tokens in memory. Algorithm 2 shows the pseudo-code for this alternative. We use the same `map` and `combine` functions as in the first phase of the BTO algorithm. Similar to BTO we use only one reducer. Figure 3.2 shows the data flow of this approach for our example dataset. The `reduce` function in OPTO gets as input a list of tokens and their partial counts. For each token, the function computes its total count and stores the information

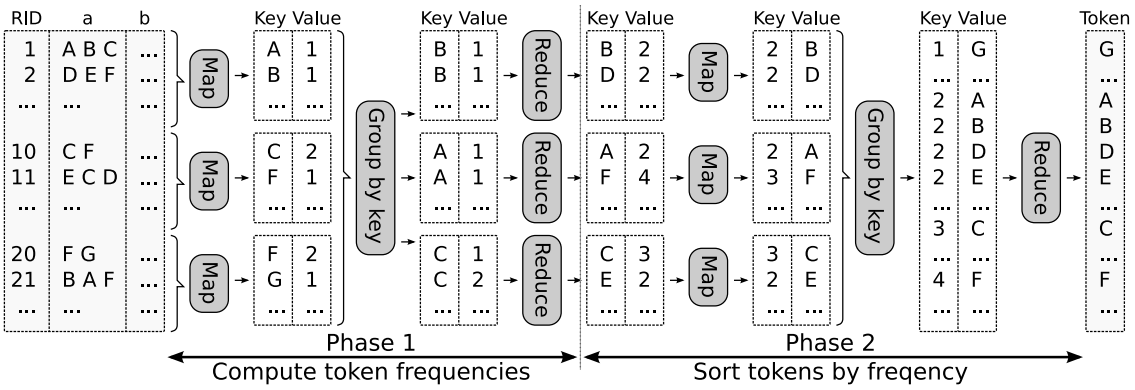


Figure 3.1: Example data flow of Stage 1 using Basic Token Ordering (BTO) for a self-join on attribute "a".

locally. When there is no more input for the `reduce` function, the reducer calls a tear-down function to sort the tokens based on their counts, and to output the tokens in increasing order based on their counts.

Algorithm 2: One-Phase Token Ordering (OPTO)

```
   /* same map and combine functions as in Basic Token Ordering,
      Phase 1                                                    */
   /* only one reduce task; the reduce function aggregates the
      counts, while the reduce_close function sorts the tokens   */
1  reduce_configure
2      token_counts ← [];
3  reduce (k2=token, list(v2)=list(partial_count))
4      total_count ← sum of partial_counts;
5      append (token, total_count) to token_counts;
6  reduce_close
7      sort token_counts by total_count;
8      foreach token in token_counts do
9          output (k3=token, v3=null);
```

3.2.2 Stage 2: RID-Pair Generation

The second stage of the join, called the "Kernel", scans the original input data and extracts the prefix of each record using the token order computed by the first stage.

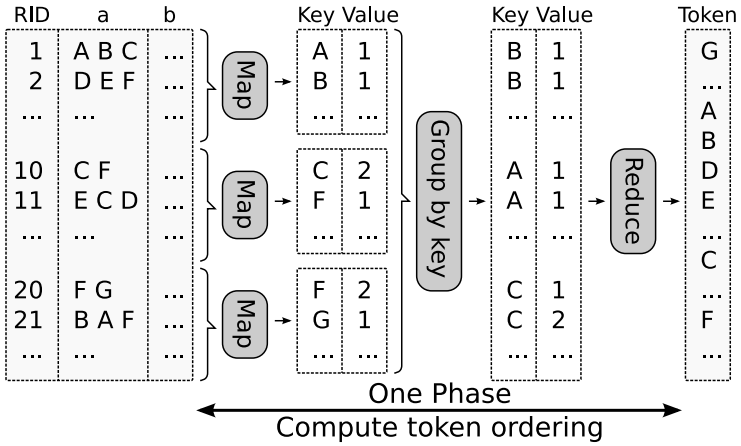


Figure 3.2: Example data flow of Stage 1 using One-Phase Token Ordering (OPTO) for a self-join on attribute "a".

In general the list of unique tokens is much smaller and grows much more slowly than the list of records. We thus assume that the list of tokens fits in memory. Based on the prefix tokens, we extract the RID and the join-attribute value of each record, and distribute these record projections to reducers. The join-attribute values that share at least one prefix token are verified at a reducer.

Routing Records to Reducers. We first take a look at two possible ways to generate `(key, value)` pairs in the `map` function. (1) *Using Individual Tokens*: This method treats each token as a key. Thus, for each record, we would generate a `(key, value)` pair for each of its prefix tokens. Thus, a record projection is replicated as many times as the number of its prefix tokens. For example, if the record value is "`A B C D`" and the prefix tokens are "`A`", "`B`", and "`C`", we would output three `(key, value)` pairs, corresponding to the three tokens. In the reducer, as the values get grouped by prefix tokens, all the values passed in a `reduce` call share the same prefix token.

(2) *Using Grouped Tokens*: This method maps multiple tokens to one synthetic key, thus can map different tokens to the same key. For each record, the `map` function generates one `(key, value)` pair for each of the groups of the prefix tokens. In our

running example of a record "A B C D", if tokens "A" and "B" belong to one group (denoted by "X"), and token "C" belongs to another group (denoted by "Y"), we output two (key, value) pairs, one for key "X" and one for key "Y". Two records that share the same token group do not necessarily share any prefix token. Continuing our running example, for record "E F G", if its prefix token "E" belongs to group "Y", then the records "A B C D" and "E F G" share token group "Y" but do not share any prefix token. So, in the reducer, as the values get grouped by their token group, no two values share a prefix token. This method can help us have fewer replications of record projections. One way to define the token groups in order to balance data across reducers is the following: We can use the token order produced in the first stage and assign the tokens to groups in a Round-Robin order. In this way, we can balance the sum of token frequencies across groups. We study the effect of the number of groups in Section 3.5. For both routing strategies, since two records might share more that one prefix token, the same pair may be verified multiple times at different reducers, and thus it could be output multiple times. This is dealt with in the third stage.

Basic Kernel (BK)

In our first approach to finding the RID pairs of similar records, called Basic Kernel ("BK"), each reducer uses a nested loop approach to compute the similarity of the join-attribute values. The pseudo-code for this alternative is shown in Algorithm 3. Before the map functions begin their executions, an initialization function is called to load the ordered tokens produced by the first stage. The map function then retrieves the original records one by one, and extracts the RID and the join-attribute value for each record. It tokenizes the join attribute and reorders the tokens based on their frequencies. Next, the function computes the prefix length and extracts the prefix tokens. Finally, the function uses either the individual tokens or the grouped tokens

routing strategy to generate the output pairs. Figure 3.3 shows the data flow for our example dataset using individual tokens to do the routing. The prefix tokens of each value are in bold face. The record with RID 1 has prefix tokens "A" and "B", so its projection is output twice.

Algorithm 3: Basic Kernel (BK)

```
1  map_configure
2      load global token order, T;
3  map (k1=unused, v1=record)
4      RID ← record ID from record;
5      A ← join attribute from record;
6      reorder tokens in A based on T;
7      compute prefix length, L, based on length(A) and similarity function and
       threshold;
8      P ← tokens in prefix of length L from A;
9      foreach token in P do
10         output (k2=token, v2=(RID, A));
11 reduce (k2=token, list(v2)=list(RID, A))
12     foreach (RID1, A1) in list(v2) do
13         foreach (RID2, A2) in list(v2) s.t. RID2 ≠ RID1 do
14             if pass_filters(A1, A2) then
15                 Sim ← similarity(A1, A2);
16                 if Sim ≥ similarity threshold then
17                     output (k3=(RID1, RID2, Sim), v3=null);
```

In the `reduce` function, for each pair of record projections, the reducer applies the additional filters (e.g., length filter, positional filter, and suffix filter) and verifies the pair if it survives. If a pair passes the similarity threshold, the reducer outputs RID pairs and their similarity values.

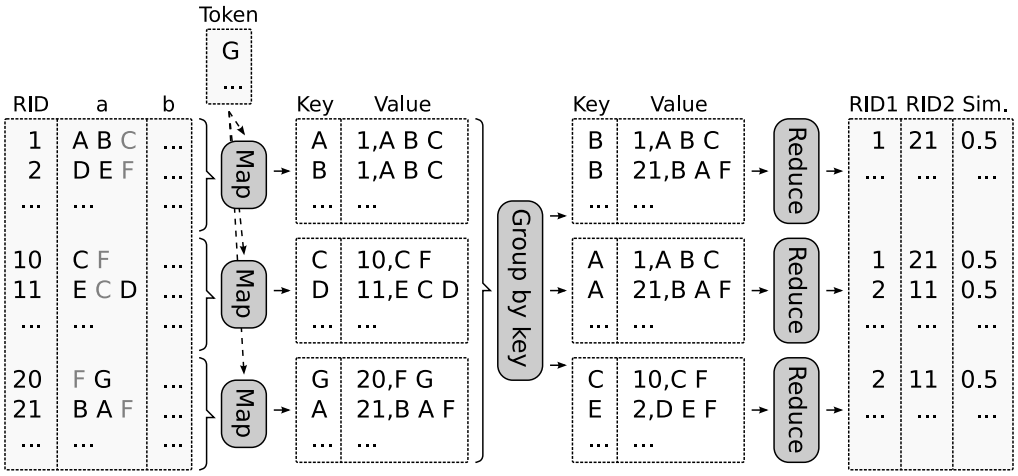


Figure 3.3: Example data flow of Stage 2 using Basic Kernel (BK), with individual tokens for routing, for a self-join on attribute "a".)

Indexed Kernel (PK)

Another approach to finding RID pairs of similar records is to use existing set-similarity join algorithms from the literature [56, 6, 10, 64]. Here we use the state-of-the-art single-machine algorithm, PPJoin+ [64]. We call this approach "PPJoin+ Kernel" ("PK").

In this method, the `map` function is the same as in the BK algorithm. Figure 3.4 shows the data flow for our example dataset using grouped tokens to do the routing. In the figure, the record with RID 1 has prefix tokens "`A`" and "`B`", which belong to groups "`X`" and "`Y`", respectively. The pseudo-code for this approach is shown in Algorithm 4. In the `reduce` function, we use the PPJoin+ algorithm to index the data, apply all the filters, and output the resulting pairs. For each input record projection, the function first probes the index using the join-attribute value. The probe generates a list of RIDs of records that are similar to the current record. The current record is then added to the index as well.

The PPJoin+ algorithm achieves an optimized memory footprint because the input

Algorithm 4: PPJoin+ Kernel (PK)

```
1  map_configure
2      load global token order, T;
3  map (k1=unused, v1=record)
       /* same computations for RID, A, and P as in lines 4-8 from
          Basic Kernel                                              */
4      G ← [] // set of unique group IDs
5      foreach token in P do
6          groupID ← choose_group(token);
7          if groupID ∉ G then
8              output (k2=groupID, v2=(RID, A));
9              insert groupID to G;
10 reduce (k2=groupID, list(v2)=list(RID, A))
11     PPJoin_init(); // initialize PPJoin index
12     foreach (RID1, A1) in list(v2) do
13         R ← PPJoin_probe(A1);
           // returns a list of (RID, Sim) pairs
14         foreach (RID2, Sim) in R do
15             output (k3=(RID1, RID2, Sim), v3=null);
16         PPJoin_insert(RID1, A1)
```

strings are sorted increasingly by their lengths [64]. This works in the following way. The index knows the lower bound on the length of the unseen data elements. Using this bound and the length filter, PPJoin+ discards from the index the data elements below the minimum length given by the filter. In order to obtain this order of data elements, we use a composite MapReduce key that also includes the length of the join-attribute value. We provide the framework with a custom partitioning function so that the partitioning is done only on the group value. In this way, when data is transferred from `map` to `reduce`, it gets partitioned just by group value, and is then locally sorted on both group and length.

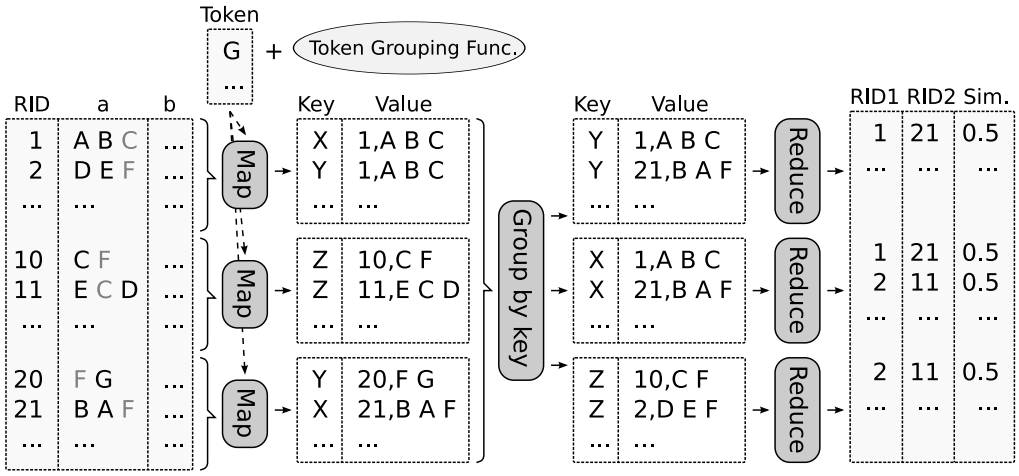


Figure 3.4: Example data flow of Stage 2 using PPJoin+ Kernel (PK), with grouped tokens for routing, for a self-join on attribute "a".

3.2.3 Stage 3: Record Join

In the final stage of our algorithm, we use the RID pairs generated in the second stage to join their records. We propose two approaches for this stage. The main idea is to first fill in the record information for each half of the pair and then use the two halves to build the full record pair. The two approaches differ in the way the list of RID pairs is provided as input. In the first approach, called Basic Record Join ("BRJ"), the list of RID pairs is treated as a normal MapReduce input, and is provided as input to the `map` functions. In the second approach, called One-Phase Record Join ("OPRJ"), the list is broadcast to all the maps and loaded before reading the input data. Duplicate RID pairs from the previous stage are eliminated in this stage.

Basic Record Join (BRJ)

The Basic Record Join algorithm uses two MapReduce phases. In the first phase, the algorithm fills in the record information for each half of each pair. In the second phase, it brings together the half-filled pairs. Algorithm 5 shows the pseudo-code for

this approach.

The `map` function in the first phase gets as input both the set of original records and the RID pairs from the second stage. (The function can differentiate between the two types of inputs by looking at the input file name.) For each original record, the function outputs a `(RID, record)` pair. For each RID pair, it outputs two `(key, value)` pairs. The first pair uses the first RID as its `key`, while the second pair uses the second RID as its `key`. Both pairs have the entire RID pair and their similarity as their value. Figure 3.5 shows the data flow for our example dataset. In the figure, the first two mappers take records as their input, while the third mapper takes RID pairs as its input. (Mappers do not span across files.) For the RID pair $(2, 11)$, the mapper outputs two pairs, one with key 2 and one with key 11.

The `reduce` function of the first phase then receives a list of values containing exactly one record and other RID pairs. For each RID pair, the function outputs a `(key, value)` pair, where the `key` is the RID pair, and the value is the record itself and the similarity of the RID pair. Continuing our example in Figure 3.5, for key 2, the first reducer gets the record with RID 2 and one RID pair $(2, 11)$, and outputs one `(key, value)` pair with the RID pair $(2, 11)$ as the key.

As an optimization for Algorithm 5, in the `reduce` of the first phase, we avoid looping twice over `list(v2)` in the following way: We use a composite key that includes a tag that indicates whether a line is a record line or a RID pair line. We then set the partitioning function so that the partitioning is done on RIDs and set the comparison function to sort the record line tags as low so that the line that contains the record will be the first element in the list.

The second phase uses an identity map that directly outputs its input. The `reduce` function therefore gets as input, for each `key` (which is a RID pair), a list of values

Algorithm 5: Basic Record Join (BRJ)

```
   // --- - Phase 1 - ---
 1 map (k1=unused, v1=line)
 2     if line is record then
 3         RID ← extract record ID from record line;
 4         output (k2=RID, v2=line);
 5     else
           // line is a (RID1, RID2, Sim) tuple
 6         output (k2=RID1, v2=line);
 7         output (k2=RID2, v2=line);

 8 reduce (k2=RID, list(v2)=list(line))
 9     foreach line in list(v2) do
10         if line is record then
11             R ← line;
12             break;
13     foreach line in list(v2) do
14         if line is a (RID1, RID2, Sim) tuple then
15             output (k3=(RID1, RID2), v3=(R, Sim));

   // --- - Phase 2 - ---
   /* identity map                                                    */
16 reduce (k2=(RID1, RID2), list(v2)=list(R, Sim))
17     R1 ← extract R from first in list(v2);
18     Sim ← extract Sim from first in list(v2);
19     R2 ← extract R from second in list(v2);
20     output (k3=(R1, Sim, R2), v3=null);
```

containing exactly two elements. Each element consists of a record and a common similarity value. The reducer forms a pair of the two records, appends their similarity, and outputs the constructed pair. In Figure 3.5, the output of the second set of mappers contains two `(key, value)` pairs with the RID pair $(1, 21)$ as the key, one containing record 1 and the other containing record 21. They are grouped in a reducer that outputs the pair of records $(1, 21)$.

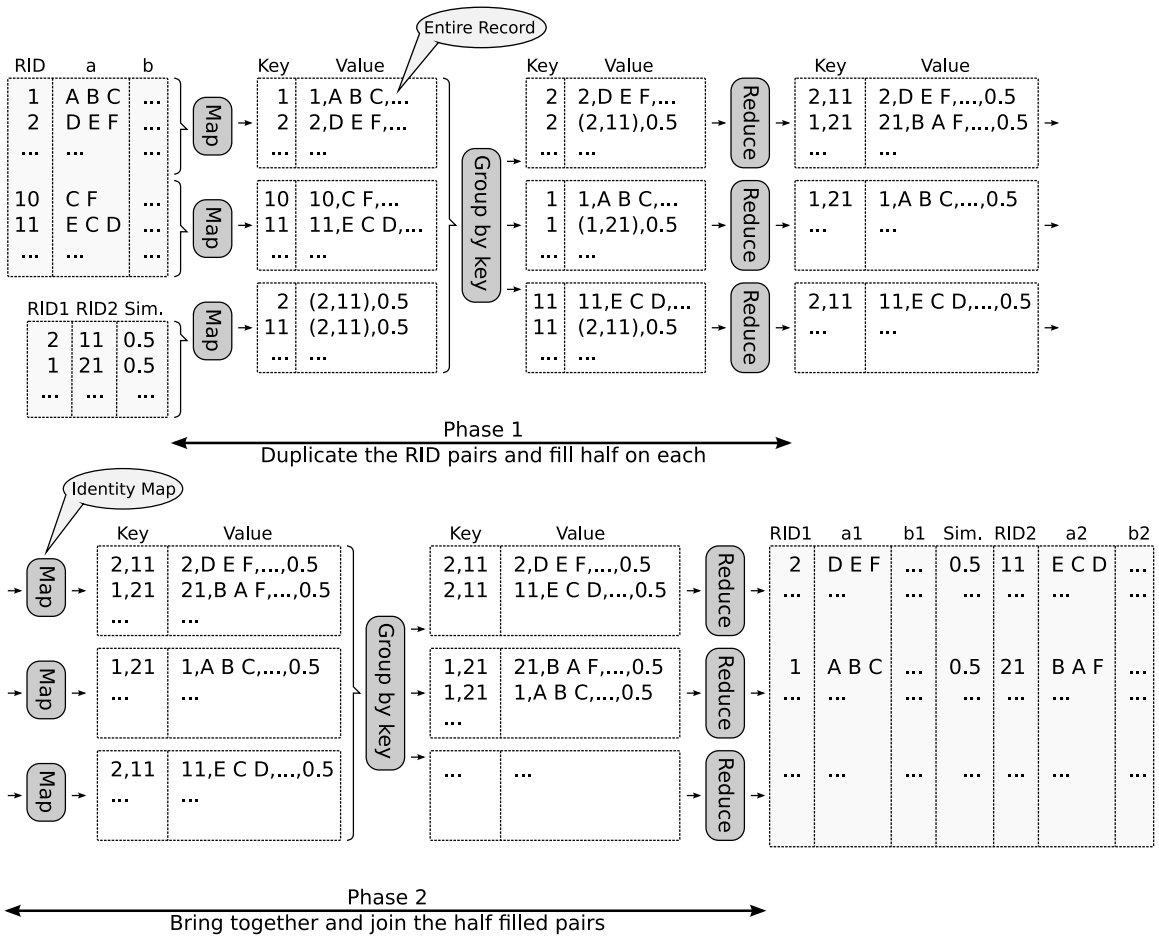


Figure 3.5: Example data flow of Stage 3 using Basic Record Join (BRJ) for a self-join case. "a1" and "a2" correspond to the original attribute "a", while "b1" and "b2" correspond to attribute "b".

One-Phase Record Join (OPRJ)

The second approach to record join uses only one MapReduce phase. Instead of sending the RID pairs through the MapReduce pipeline to group them with the records in the reduce phase (as we do in the BRJ approach), we broadcast and load the RID pairs at each `map` function before the input data is consumed by the function. Algorithm 6 shows the pseudo-code for the second approach. The `map` function then gets the original records as input. For each record, the function outputs as many `(key, value)` pairs as the number of RID pairs containing the RID of the current record. The output `key` is the RID pair. Essentially, the output of the `map` function is

the same as the output of the `reduce` function in the first phase of the BRJ algorithm. The idea of joining the data in the mappers was also used in [29] for the case of equi-joins. The `reduce` function is the same as the `reduce` function in the second phase of the BRJ algorithm. Figure 3.6 shows the data flow for our example dataset. In the figure, the first mapper gets as input the record with RID 1 and outputs one `(key, value)` pair, where the key is the RID pair $(1, 21)$ and the value is record 1. On the other hand, the third mapper outputs a pair with the same key, and the value is the record 21. The two pairs get grouped in the second reducer, where the pair of records $(1, 21)$ is output.

Algorithm 6: One-Phase Record Join (OPRJ)

```
1 map_configure
2 |  load RID pairs and build a hash table, P, with the format RID1 → {(RID2,
  |  Sim), ...};
3 map (k1=unused, v1=record)
4 |  extract RID1 from record;
5 |  J ← probe P for RID1;
6 |  foreach (RID2, Sim) in J do
7 |  |  output (k3=(RID1, RID2), v3=(record, Sim));
  /* same reduce function as in Basic Data Join, Phase 2          */
```

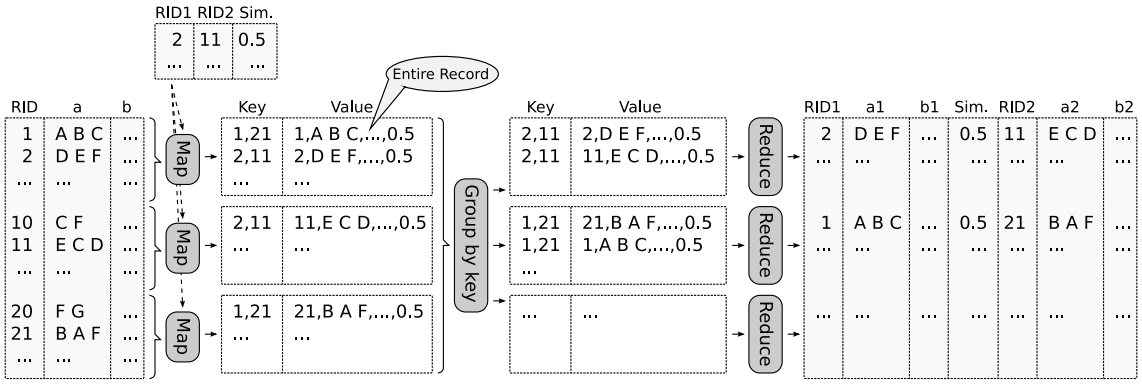


Figure 3.6: Example data flow of Stage 3 using One-Phase Record Join (OPRJ) for a self-join case. "a1" and "a2" correspond to the original attribute "a" while, "b1" and "b2" correspond to attribute "b".

3.3 R-S Join Case

In Section 3.2 we described how to compute set-similarity self-joins using the MapReduce framework. In this section we present our solutions for the set-similarity R-S join case. We highlight the differences between the two cases and discuss an optimization for carefully controlling memory usage in the second stage.

The main differences between the two join cases lie in the second and the third stages, where we have records from two datasets as the input. Dealing with the binary join operator is challenging in MapReduce, as the framework was designed to operate on a single input stream. As discussed in [29, 53], in order to differentiate between two different input streams in MapReduce, we can extend the key of the `(key, value)` pairs so that it includes a relation *tag* for each record. We can also modify the partitioning function so that partitioning is done on the part of the key that does not include the relation name. (However, the sorting is still done on the full key.) We now explain the three stages of an R-S join.

Stage 1: Token Ordering. In the first stage, we use the same algorithms as in the self-join case, only on the relation with fewer records, say R. In the second stage, when tokenizing the other relation, S, we discard the tokens that do not appear in the token list, since they cannot generate candidate pairs with R records.

Stage 2: Basic Kernel. First, the mappers tag the record projections with their relation name. Thus, the reducers receive a list of record projections grouped by relation. In the `reduce` function, we then store the records from the first relation (as they arrive first), and stream the records from the second relation (as they arrive later). For each record in the second relation, we verify it against all the records in the first relation.

Stage 2: Indexed Kernel. We use the same mappers as for the Basic Kernel. The reducers index the record projections of the first relation and probe the index for the record projections of the second relation.

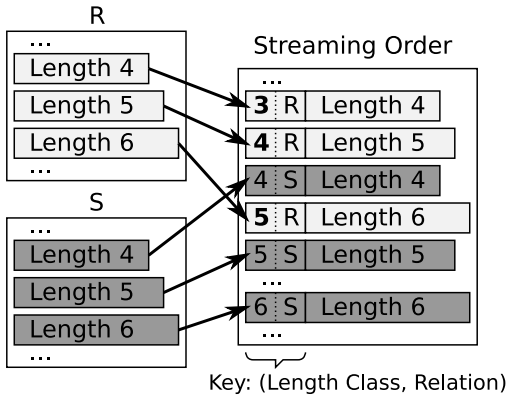


Figure 3.7: Example of the order in which records need to arrive at the reducer in the PK kernel of the R-S join case, assuming that for each length, l, the lower-bound is $l-1$ and the upper-bound is $l+1$.

As in the self-join case, we can minimize the memory footprint of the `reduce` function by having the data sorted increasingly by their lengths. PPJoin+ only considered this improvement for self-joins. For R-S joins, the challenge is that we need to make sure that we first stream all the record projections from R that might join with a particular S record before we stream this record. Specifically, given the length of a set, we can define a lower-bound and an upper-bound on the lengths of the sets that might join with it [6]. Before we stream a particular record projection from S, we need to have seen all the record projections from R with a length smaller than or equal to the *upper-bound* length of the record from S. We force this arrival order by extending the keys with a *length class* assigned in the following way. For records from S, the length class is their actual length. For records from R, the length class is the *lower-bound* length corresponding to their length. Figure 3.7 shows an example of the order in which records will arrive at the reducer, assuming that for each length l, the lower-bound is $l-1$ and the upper-bound is $l+1$. In the figure, the records from R with length 5 get length class 4 and are streamed to the reducer before those records from

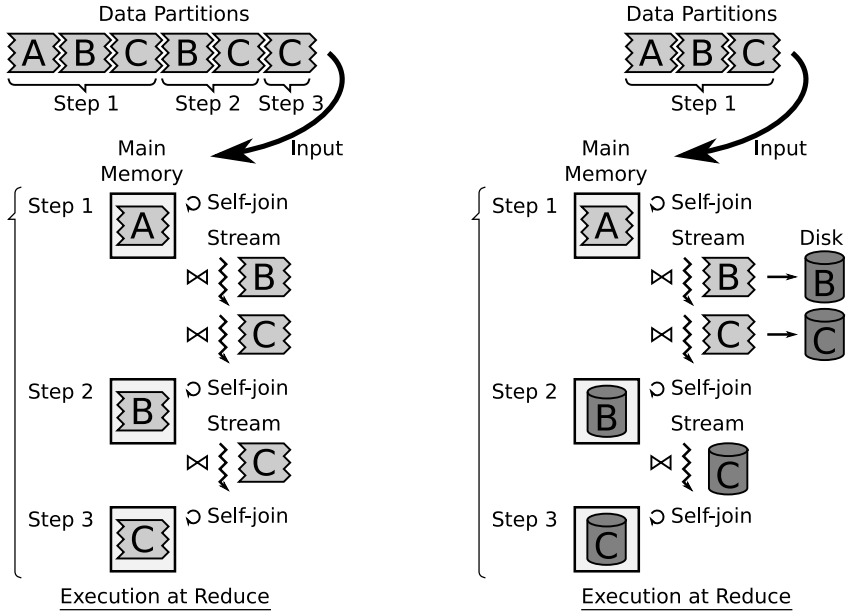
S with lengths between $[4, 6]$.

Stage 3: Record Join. For the BRJ algorithm the mappers first tag their outputs with the relation name. Then, the reducers get a record and their corresponding RID pairs grouped by relation and output half-filled pairs tagged with the relation name. Finally, the second-phase reducers use the relation name to build record pairs having the record form R first and the record form S second. In the OPRJ algorithm, for each input record from R, the mappers output as many `(key,value)` pairs as the number of RID pairs containing the record's RID in the R column (and similar for S records). For each pair, the key is the RID pair plus the relation name. The reducers proceed as do the second-phase reducers for the BRJ algorithm.

3.4 Handling Insufficient Memory

As we saw in Section 3.2.2, reducers in the second stage receive as input a list of record projections to be verified. In the BK approach, the entire list of projections needs to fit in memory. (For the R-S join case, only the projections of one relation must fit.) In the PK approach, because we are exploiting the length filter, only the fragment corresponding to a certain length range needs to fit in memory. (For the R-S join case, only the fragment belonging to only one relation must fit.) It is worth noting that we already decreased the amount of memory needed by grouping the records on the infrequent prefix tokens. Moreover, we can exploit the length filter even in the BK algorithm, by using the length filter as a secondary record-routing criterion. In this way, records are routed on token-length-based keys. The additional routing criterion partitions the data even further, decreasing the amount of data that needs to fit in memory. This technique can be generalized and additional filters can be added to the routing criteria. In this section we present two extensions of our

algorithms for the case where there are just no more filters to be used but the data still does not fit in memory. The challenge is how to compute the cross product of a list of elements in MapReduce. We sub-partition the data so that each block fits in memory and propose two approaches for processing the blocks. First we look how the two methods work in the self-join case and then discuss the differences for the R-S join case.



(a) Map-based block processing

(b) Reduce-based block processing

Figure 3.8: Data flow in the reducer for two block processing approaches.

Map-Based Block Processing. In this approach, the `map` function replicates the blocks and interleaves them in the order they will be processed by the reducer. For each block sent by the `map` function, the reducer either loads the block in memory or streams the block against a block already loaded in memory. Figure 3.8(a) shows an example of how blocks are processed in the reducer, in which the data is sub-partitioned into three blocks A, B, and C. In the first step, the first block, A, is loaded into memory and self-joined. After that, the next two blocks, B and C, are read from the input stream and joined with A. Finally, A is discarded from memory

and the process continues for blocks B and C. In order to achieve the interleaving and replications of the blocks, the `map` function does the following. For each `(key, value)` output pair, the function determines the pair's block, and outputs the pair as many times as the block needs to be replicated. Every copy is output with a different composite key, which includes its position in the stream, so that after sorting the pairs, they are in the right blocks and the blocks are in the right order.

Reduce-Based Block Processing. In this approach, the `map` function sends each block exactly once. On the other hand, the `reduce` function needs to store all the blocks except the first one on its local disk, and reload the blocks later from the disk for joining. Figure 3.8(b) shows an example of how blocks are processed in the reducer for the same three blocks A, B, and C. In the first step, block A is loaded into memory and self-joined. After that, the next two blocks, B and C, are read from the input stream and joined with A and also stored on the local disk. In the second step, A is discarded from memory and block B is read from disk and self-joined. Then, block C is read from the disk and joined with B. The process ends with reading C from disk and self-joining it.

Handling R-S Joins. In the R-S join case the `reduce` function needs to deal with a partition from R that does not fit in memory, while it streams a partition coming from S. We only need to sub-partition the R partition. The `reduce` function loads one block from R into memory and streams the entire S partition against it. In the map-based block processing approach, the blocks from R are interleaved with multiple copies of the S partition. In the reduce-based block processing approach all the R blocks (except the first one) and the entire S partition are stored and read from the local disk later.

3.5 Experimental Evaluation

In this section we describe the performance evaluation of the proposed algorithms. To understand the performance of the parallel algorithms we measured their absolute running time (wall clock) as well as their speedup and scaleup [26].

We ran experiments on a 10-node IBM x3650 cluster. Each node had one Intel Xeon processor E5520 2.26GHz with four cores, 12GB of RAM, and four 300GB hard disks. Thus the cluster consists of 40 cores and 40 disks. We used an extra node for running the master daemons to manage the Hadoop jobs and the Hadoop distributed file system. On each node, we installed the Ubuntu 9.04, 64-bit, server edition operating system, Java 1.6 with a 64-bit server JVM, and Hadoop 0.20.1. In order to maximize the parallelism and minimize the running time, we made the following changes to the default Hadoop configuration: we set the block size of the distributed file system to 128MB, allocated 1GB of virtual memory to each daemon and 2.5GB of virtual memory to each map/reduce task, ran four map and four reduce tasks in parallel on each node, set the replication factor to 1, and disabled the speculative task execution feature.

We used the following two datasets and increased their sizes as needed:

- **DBLP:**[1] It had about 1.2 million publications. We preprocessed the original XML file by removing the tags, and output one line per publication that contained a unique integer (RID), a title, a list of authors, and the rest of the content (publication date, publication journal or conference, and publication medium). The average length of a record was 259 bytes. A copy of the entire dataset has around 300MB before we increased its size. It is worth noting that we did not *clean* the records before running our algorithms, i.e., we did not

[1] http://dblp.uni-trier.de/xml/dblp.xml.gz

remove punctuations or change the letter cases. We did the cleaning inside our algorithms.

- **CITESEERX:**[2] It had about 1.3 million publications. We preprocessed the original XML file in the same way as for the DBLP dataset. Each publication included an abstract and URLs to its references. The average length of a record was 1374 bytes, and the size of one copy of the entire dataset is around 1.8GB.

Increasing Dataset Sizes. To evaluate our parallel set-similarity join algorithms on large datasets, we increased each dataset while maintaining its set-similarity join properties. We maintained a roughly constant token dictionary, and wanted the cardinality of join results to increase linearly with the increase of the dataset. Increasing the data size by duplicating its original records would only preserve the token-dictionary size, but would blow up the size of the join result. To achieve the goal, we increased the size of each dataset by generating new records as follows. We first computed the frequencies of the tokens appearing in the title and the list of authors in the original dataset, and sorted the tokens in their increasing order of frequencies. For each record in the original dataset, we created a new record by replacing each token in the title or the list of authors with the token after it in the token order. For example, if the token order is `(A, B, C, D, E, F)` and the original record is "`B A C E`", then the new record is "`C B D F`." We evaluated the cardinality of the join result after increasing the dataset in this manner, and noticed it indeed increased linearly with the increase in the dataset size.

We increased the size of each dataset 5 to 25 times. We refer to the increased datasets as "DBLP$\times n$" or "CITESEERX$\times n$", where $n \in [5, 25]$ and represents the increase factor. For example, "DBLP$\times$5" represents the DBLP dataset increased five times.

[2]http://citeseerx.ist.psu.edu/about/metadata

Before starting each experiment we balanced its input datasets across the ten nodes in HDFS and the four hard drives of each node in the following way. We formatted the distributed file system before each experiment. We created an identity MapReduce job with as many reducers running in parallel as the number of hard disks in the cluster. We exploited the fact that reducers write their output data to the local node and also the fact that Hadoop chooses the disk to write the data using a Round-Robin order.

For all the experiments, we tokenized the data by word. We used the concatenation of the paper title and the list of authors as the join attribute, and used the Jaccard similarity function with a similarity threshold of 0.80. The 0.80 threshold is usually the lower bound on the similarity threshold used in the literature [6, 18, 64], and higher similarity thresholds decreased the running time. The source code is available at `http://asterix.ics.uci.edu/fuzzyjoin`.

3.5.1 Self-Join Performance

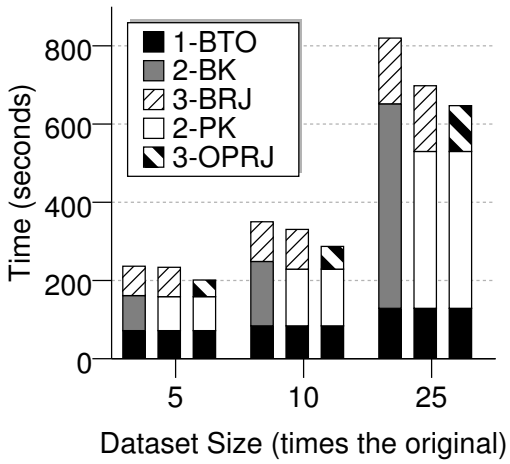


Figure 3.9: Running time for self-joining DBLP$\times n$ datasets (where $n \in [5, 25]$) on the 10-node cluster.

We did a self-join on the DBLP$\times n$ datasets, where $n \in [5, 25]$. Figure 3.9 shows

the total running time of the three stages on the 10-node cluster for different dataset sizes (represented by the factor n). The running time consisted of the times of the three stages: token ordering, kernel, and record join. For each dataset size, we used three combinations of the approaches of the stages. For example, "1-BTO" means we use BTO in the first stage. The second stage is the most expensive step, and its time increased the fastest with the increase of the dataset size. The best algorithm is BTO-PK-OPRJ, i.e., with BTO for the first stage, PK for the second stage, and OPRJ for the third stage. This combination could self-join 25 times the original DBLP dataset in around 650 seconds.

Self-Join Speedup

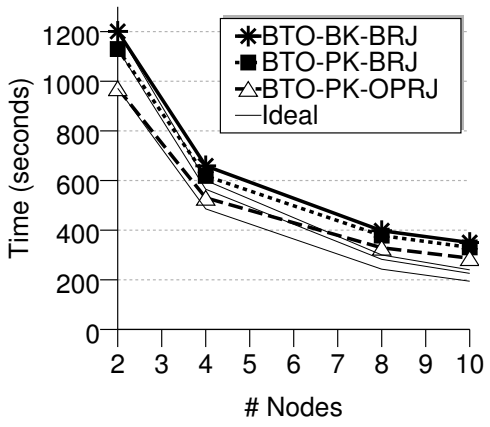


Figure 3.10: Running time for self-joining the DBLP×10 dataset on different cluster sizes.

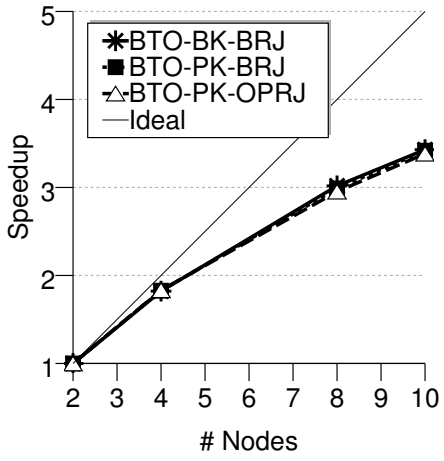


Figure 3.11: Relative running time for self-joining the DBLP×10 data set on different cluster sizes.

In order to evaluate the speedup of the approaches, we fixed the dataset size and varied the cluster size. Figure 3.10 shows the running time for self-joining the DBLP×10 dataset on clusters of 2 to 10 nodes. We used the same three combinations for the three stages. For each approach, we also show its ideal speedup curve (with a thin black line). For instance, if the cluster has twice as many nodes and the data size

does not change, the approach should be twice as fast. In Figure 3.11 we show the same numbers, but plotted on a "relative scale". That is, for each cluster size, we plot the ratio between the running time for the smallest cluster size and the running time of the current cluster size. For example, for the 10-node cluster, we plot the ratio between the running time on the 2-node cluster and the running time on the 10-node cluster. We can see that all three combinations have similar speedup curves, but none of them speed up linearly. In all the settings the BTO-PK-OPRJ combination is the fastest (Figure 3.10). In the following experiments we looked at the speedup characteristics for each of the three stages individually in order to better understand the overall speedup.

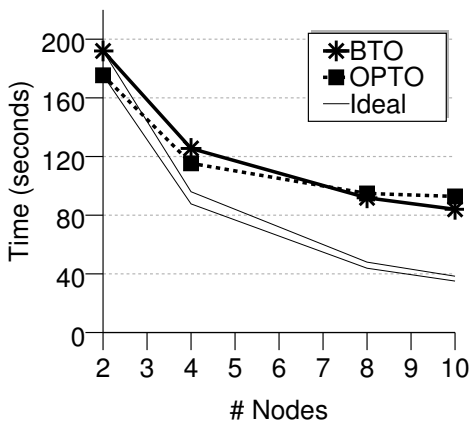


Figure 3.12: Running time of Stage 1 (token ordering) for self-joining the DBLP×10 dataset on different cluster sizes.

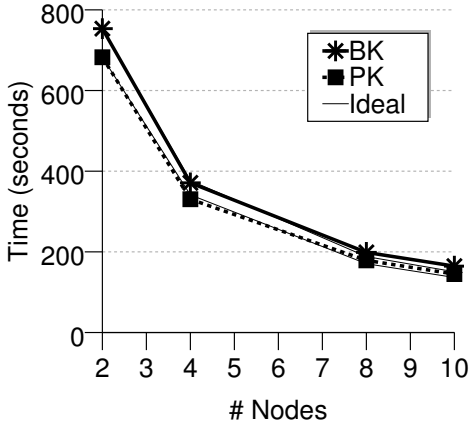


Figure 3.13: Running time of Stage 2 (kernel) for self-joining the DBLP×10 dataset on different cluster sizes.

Stage 1: Token Ordering. Figure 3.12 shows the running time for the first stage of the self-join (token ordering). We can see that the OPTO approach was the fastest for the settings of 2 nodes and 4 nodes. For the settings of 8 nodes and 10 nodes, the BTO approach became the fastest. Their limited speedup was due to two main reasons. (1) As the number of nodes increased, the amount of input data fed to each combiner decreased. As the number of nodes increased, more data was sent

through the network and more data got merged and reduced. (A similar pattern was observed in [29] for the case of general aggregations.) (2) The final token ordering was produced by only one reducer, and this step's cost remained constant as the number of nodes increased. The speedup of the OPTO approach was even worse since as the number of nodes increased, the extra data that was sent through the network had to be aggregated at only one reducer. Because BTO was the fastest for settings of 8 nodes and 10 nodes, and it sped up better than OPTO, we only considered BTO for the end-to-end combinations.

Stage 2: Kernel. In Figure 3.13 we plotted the running time for the second stage (kernel) of the self-join. For the PK approach, an important factor affecting the running time is the number of token groups. We evaluated the running time for different numbers of groups. We observed that the best performance was achieved when there was one group per token. The reason was that the `reduce` function could benefit from the grouping conducted "for free" by the MapReduce framework. If groups had more than one token, the framework spends the same amount of time on grouping, but the reducer benefits less. Both approaches had an almost perfect speedup. Moreover, in all the settings, the PK approach was the fastest.

Stage 3: Record Join. Figure 3.14 shows the running time for the third stage (record join) of the self-join. The OPRJ approach was always faster than the BRJ approach. The main reason for the poor speedup of the BRJ approach was due to the skew in the RID pairs that join, which affected the workload balance. For analysis purposes, we computed the frequency of each RID appearing in at least one RID pair. On the average an RID appeared on 3.74 RID pairs, with a standard deviation of 14.85 and a maximum of 187. Additionally, we counted how many records were processed by each `reduce` instance. The minimum number of records processed in the 10-nodes case was 81,662 and the maximum was 90,560, with an average of 87,166.55

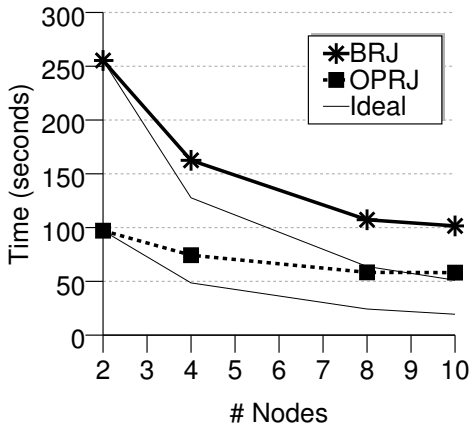


Figure 3.14: Running time of Stage 3 (record join) for self-joining the DBLP×10 dataset on different cluster sizes.

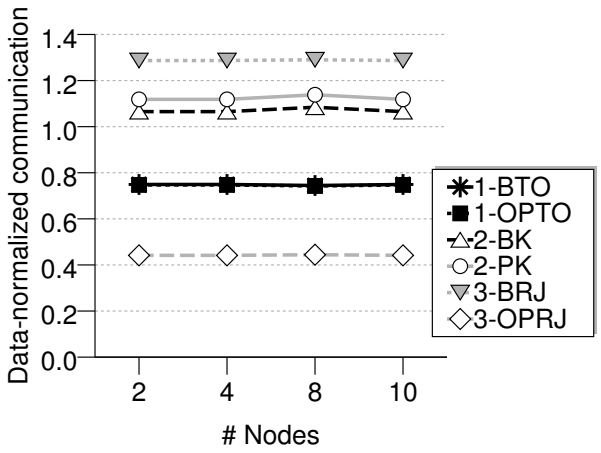


Figure 3.15: Data-normalized communication in each stage for self-joining the DBLP×10 dataset on different cluster sizes.

and a standard deviation of 2,519.30. No matter how many nodes we added to the cluster, a single RID could not be processed by more than one `reduce` instance, and all the reducers had to wait for the slowest one to finish.

The speedup of the OPRJ approach was limited because in the OPRJ approach, the list of RID pairs that joined was broadcast to all the maps where they must be loaded in memory and indexed. The elapsed time required for this remained constant as the number of nodes increased.

Figure 3.15 shows the data-normalized communication in each stage. Each point shows the ratio between the total size of the data sent between `map` and `reduce` and the size of the original input data. Note that only a fraction of that data is transferred through the network to other nodes while the rest is processed locally.

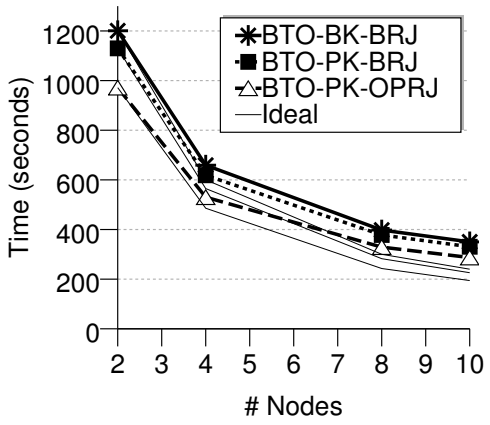


Figure 3.16: Running time for self-joining the DBLP×n dataset (where $n \in [5, 25]$) increased proportionally with the increase of the cluster size.

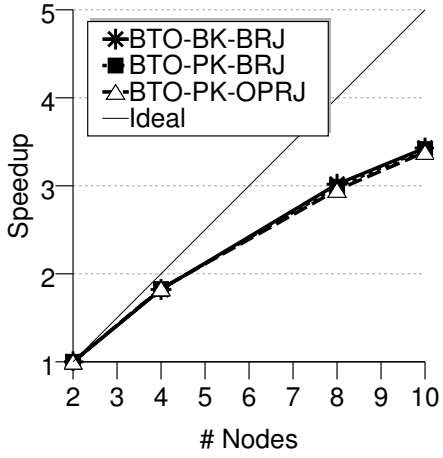


Figure 3.17: Relative running time for self-joining the DBLP×n data set (where $n \in [5, 25]$) increased proportionally with the increase of the cluster size.

Self-Join Scaleup

In order to evaluate the scaleup of the proposed approaches we increased the dataset size and the cluster size together by the same factor. A perfect scaleup could be achieved if the running time remained constant. Figure 3.16 shows the running time for self-joining the DBLP dataset, increased from 5 to 25 times, on a cluster with 2 to 10 nodes, respectively. Figure 3.17 shows the running time for self-joining the DBLP dataset, increased from 5 to 25 times, on a cluster with 2 to 10 nodes, respectively on a "relative" scale. For each dataset size we plotted the ratio between the running time for that dataset size and the running time for the minimum dataset size. We can see that the fastest combined algorithm was BTO-PK-OPRJ. We can also see that all three combinations scaled up well. BTO-PK-BRJ had the best scaleup. In the following, we look at the scaleup characteristics of each stage.

Stage 1: Token Ordering. Figure 3.18 shows the running time of the first stage (token ordering). We can see that the BTO approach scaled up almost perfectly,

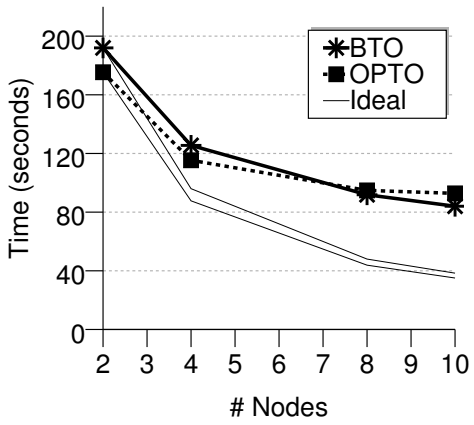


Figure 3.18: Running time of Stage 1 (token ordering) for self-joining the DBLP×n ($n \in [5, 25]$) dataset increased proportionally with the increase of the cluster size.

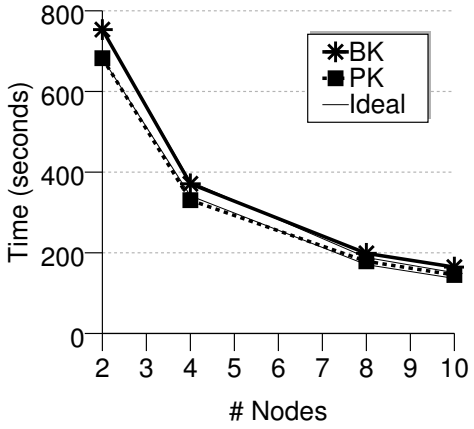


Figure 3.19: Running time of Stage 2 (kernel) for self-joining the DBLP×n ($n \in [5, 25]$) dataset increased proportionally with the increase of the cluster size.

while the OPTO approach did not scale up as well. Moreover, the OPTO approach became more expensive than the BTO approach as the number of nodes increased. The reason why the OPTO approach did not scale up as well was because it used only one `reduce` instance to aggregate the token counts (instead of using multiple `reduce` functions as in the BTO case). As the data increased, the time needed to finish the one `reduce` function increased. Both approaches used a single `reduce` to sort the tokens by frequency.

Stage 2: Kernel. The running time for the second stage (kernel) is plotted in Figure 3.19. We can see that the PK approach was always faster and scaled up better than the BK approach. To understand why the BK approach did not scale up well, let us take a look at the time complexity of the reducers. The `reduce` function was called for each prefix token. For each token, the `reduce` function received a list of record projections, and for BK it had to verify the self-cross-product of this list. Thus, if the length of the record projections list is n and the number of tokens that each BK reducer has to process is m, the time complexity of each reducer is $O(m \cdot n^2)$.

As the dataset increases, the number of unique tokens remains constant, but the number of records having a particular prefix token increased by the same factor as the dataset size. Thus, when the dataset is increased t times, the length of the record projections list also increases t times. Moreover, as the number of nodes increases t times, the number of tokens that each reducer has to process decreases t times. Thus, the complexity of each reducer becomes $O\big(m/t \cdot (n \cdot t)^2\big) = O(t \cdot m \cdot n^2)$. Notice that despite the fact the `reduce` function had a running time that grew proportional with the dataset size, the overall scaleup of the BK approach was still acceptable because its `map` function scaled up well. In the case of PK, the quadratic reducer increase when the data linearly increased was alleviated because an index was used to decide which pairs are verified.

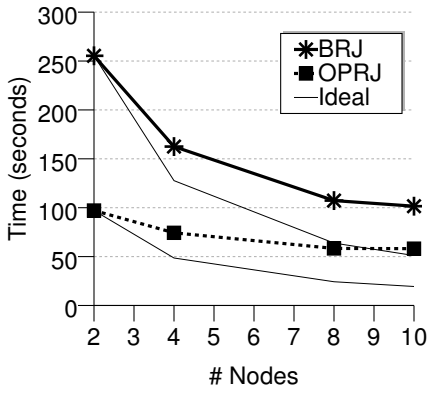


Figure 3.20: Running time of Stage 3 (record join) for self-joining the DBLP$\times n$ ($n \in [5, 25]$) dataset increased proportionally with the increase of the cluster size.

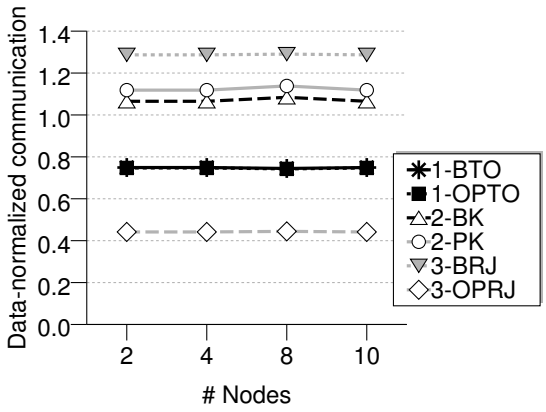


Figure 3.21: Data-normalized communication in each stage for self-joining the DBLP$\times n$ dataset ($n \in [5, 25]$) increased proportionally with the increase of the cluster size.

Stage 3: Record Join. Figure 3.20 shows the running time for the third stage (record join). We can see that the BRJ approach had an almost perfect scaleup, while the OPRJ approach did not scale up well. For our 10-node cluster, the OPRJ approach was faster than the BRJ approach, but OPRJ could become slower as the number of nodes and data size increased. The OPRJ approach did not scale up well

since the list of RID pairs that needed to be loaded and indexed by each `map` function increased linearly with the size of the dataset. Figure 3.21 shows the data-normalized communication in each stage.

Self-Join Performance Summary

We have the following observations:

- For the first stage, BTO was the best choice.
- For the second stage, PK was the best choice.
- For the third stage, the best choice depends on the amount of data and the size of the cluster. In our experiments, OPRJ was somewhat faster, but the cost of loading the similar-RID pairs in memory was constant as the the cluster size increased, and the cost increased as the data size increased. For these reasons, we recommend BRJ as a good alternative.
- The three combinations had similar speedups, but the best scaleup was achieved by BTO-PK-BRJ.
- Our algorithms distributed the data well in the first and the second stages. For the third stage, the algorithms were negatively affected by the fact that some records produced more join results that others, and the amount of work to be done was not well balanced across nodes. This skew heavily depends on the characteristics of the data and we propose to study this issue in future work.
- Data replication based on their prefix tokens did not have a significant negative effect on the performance of our algorithms.

3.5.2 R-S-Join Performance

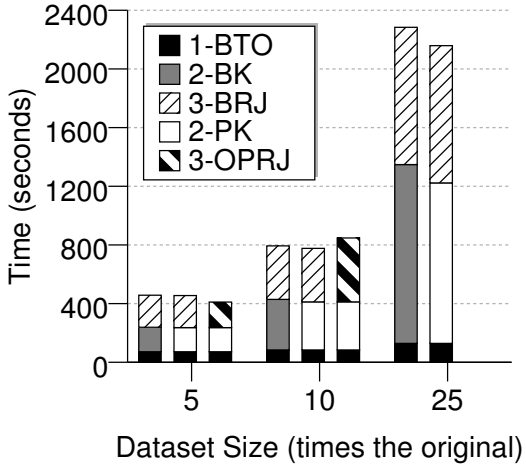


Figure 3.22: Running time for joining the DBLP$\times n$ and the CITESEERX$\times n$ datasets (where $n \in [5, 25]$) on a 10-node cluster.

To evaluate the performance of the algorithms for the R-S-join case, we did a join between the DBLP and the CITESEERX datasets. We increased both datasets at the same time by a factor between 5 and 25. Figure 3.22 shows the running time for the join on the 10-node cluster. We used the same set of combinations for each stage as in the self-join case. Moreover, the first stage was identical to the first stage of the self-join case, as this stage was run on only one of the datasets, in this case, DBLP. The running time for the second stage (kernel) increased the fastest compared with the other stages, but for the 5 and 10 dataset-increase factors, the third stage (record join) became the most expensive. The main reason for this behavior, compared to the self-join case, was that this stage had to scan two datasets instead of one, and the record length of the CITESEERX dataset was much larger than the record length of the DBLP dataset. For the 25 dataset-increase factor, the OPRJ approach ran out of memory when it loaded the list of RID pairs, making BRJ the only option.

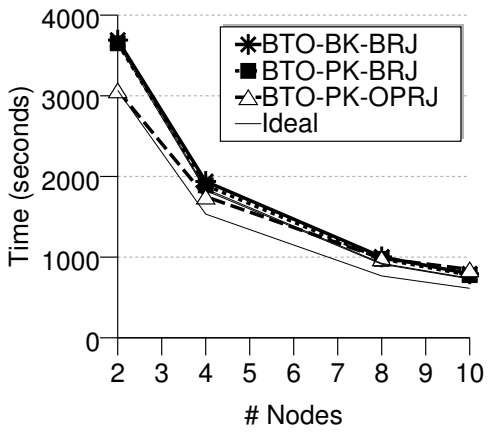


Figure 3.23: Running time for joining the DBLP×10 and the CITESEERX×10 datasets on different cluster sizes.

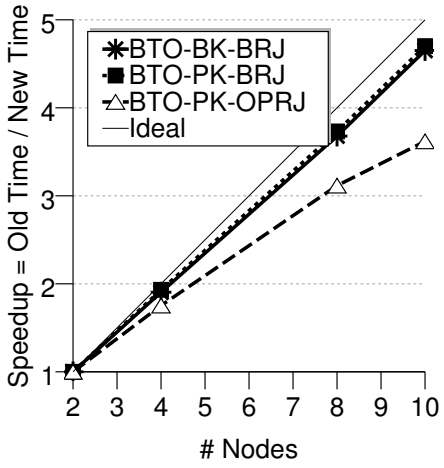


Figure 3.24: Relative running time for joining the DBLP×10 and the CITESEERX×10 datasets on different cluster sizes.

R-S-Join Speedup

As in the self-join case, we evaluated the speedup of the algorithms by keeping the dataset size constant and increasing the number of nodes in the cluster. Figure 3.23 shows the running times for the same three combinations of approaches. Figure 3.24 shows the relative running time for joining the DBLP×10 and the CITESEERX×10 datasets on clusters of 2 to 10 nodes. We used the same three combinations for the three stages. We also show the ideal-speedup curve (with a thin black line). We can see that the BTO-PK-OPRJ combination was initially the fastest, but for the 10-node cluster, it became slightly slower than the BTO-BK-BRJ and BTO-PK-BRJ combinations. Moreover, BTO-BK-BRJ and BTO-PK-BRJ sped up better than BTO-PK-OPRJ.

To better understand the speedup behavior, we looked at each individual stage. The first stage performance was identical to the first stage in the self-join case. Figure 3.25 shows the running time for the second stage of the join (kernel). For this stage, we

noticed a similar speedup (almost perfect) as for the self-join case. In Figure 3.26 we plot the running time for the third stage (record join) of the join. Regarding this, we noticed that the OPRJ approach was initially the fastest (for the 2 and 4 node case), but it eventually became slower than the BRJ approach. Additionally, the BRJ approach sped up better than the OPRJ approach. The poor performance of the OPRJ approach was due to the fact that all the map instances had to load the list of RID pairs that join.

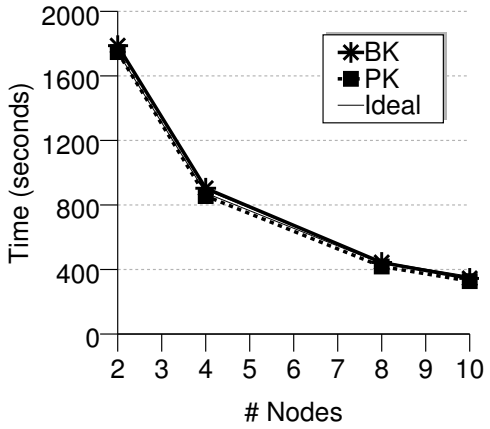


Figure 3.25: Running time of Stage 2 (kernel) for joining the DBLP×10 and the CITESEERX×10 datasets on different cluster sizes.

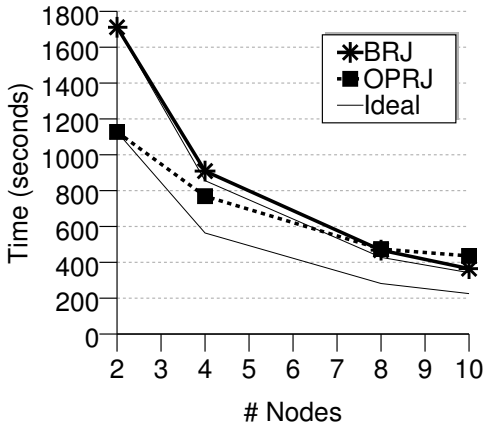


Figure 3.26: Running time of Stage 3 (record join) for joining the DBLP×10 and the CITESEERX×10 datasets on different cluster sizes.

R-S-Join Scaleup

We also evaluated the scaleup of the R-S-join approaches. The evaluation was similar to the one done in the self-join case. In Figure 3.27 we plot the running time of three combinations for the three stages as we increased the dataset size and the cluster size by the same factor. Figure 3.28 shows the running time for joining the DBLP and the CITESEERX datasets, increased from 5 to 25 times, on a cluster with 2 to 10 nodes, respectively on a "relative" scale. We can see that BTO-BK-BRJ and BTO-PK-BRJ scaled up well. The BTO-PK-BRJ combination scaled up the best. BTO-PK-OPRJ

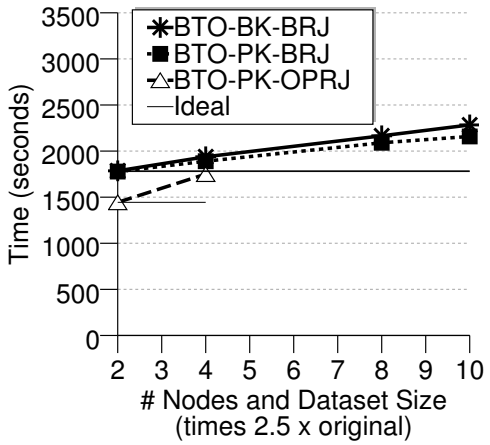


Figure 3.27: Running time for joining the DBLP×n and the CITESEERX×n datasets (where $n \in [5, 25]$) increased proportionally with the cluster size.

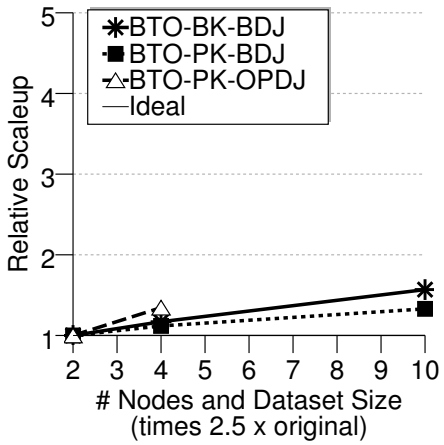


Figure 3.28: Relative running time for joining the DBLP×n and the CITESEERX×n datasets (where $n \in [5, 25]$) increased proportionally with the cluster size.

ran out of memory in the third stage for the case where the datasets were increased 8 times the original size. The third stage ran out of memory when it tried to load in memory the list of RID pairs that join. Before running out of memory, though, BTO-PK-OPRJ was the fastest.

To better understand the behavior of our approaches, we again analyzed the scaleup of each individual stage. The first stage performance was identical with its counterpart in the self-join case. Figure 3.29 shows the running time of the second stage (kernel). We noticed similar scaleup performance as its counterpart in the self-join case. The running time for the third stage (record join) is plotted in Figure 3.30. Regarding this, we observed that the BRJ approach scaled up well. We also observed that even before running out of memory, the OPRJ approach did not scale up well, but for the case where it did not run out of memory, it was faster than the BRJ approach.

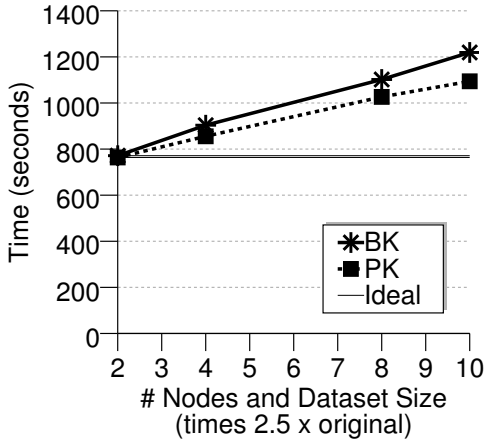


Figure 3.29: Running time of Stage 2 (kernel) for joining the DBLP$\times n$ and the CITESEERX$\times n$ datasets (where $n \in [5, 25]$) increased proportionally with the increase of the cluster size.

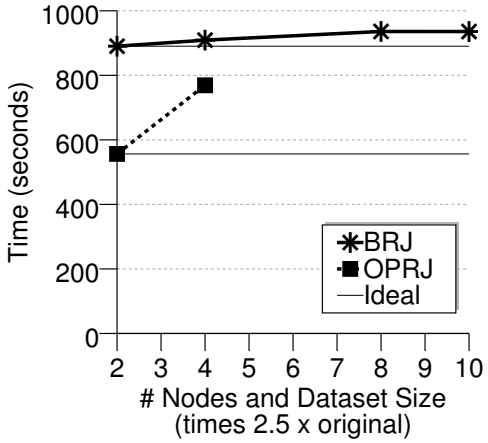


Figure 3.30: Running time of Stage 3 (record join) for joining the DBLP$\times n$ and the CITESEERX$\times n$ datasets (where $n \in [5, 25]$) increased proportionally with the increase of the cluster size.

R-S-Join Performance Summary

We have the following observations:

- The recommendations for the best choice from the self-join case also hold for the R-S-join case.
- The third stage of the join became a more significant part of the execution due to the fact that we have two datasets.
- The three algorithm combinations preserved their speedup and scaleup characteristics as for the self-join case.
- We also observed the same workload-balancing characteristics as for the self-join case.
- For both the self-join and R-S-join cases, we recommend the combination BTO-PK-BRJ as a robust and scalable overall method.

3.6 Conclusions

In this chapter we have studied the problem of answering set-similarity join queries in parallel using the MapReduce framework. We proposed a three-stage approach and explored several solutions for each stage. We showed how to partition the data across nodes in order to balance the workload and minimize the need for replication. We discussed ways to efficiently deal with partitioning, replication, and multiple inputs by exploiting the characteristics of the MapReduce framework. We also described how to control the amount of data that needs to be kept in memory during join by exploiting the data properties. We studied both self-joins and R-S joins, end-to-end, by starting from complete records and producing complete record pairs. Moreover, we discussed strategies for dealing with extreme situations where, even after the data is partitioned to the finest granularity, the amount of data that needs to be in the main memory of one node is too large to fit. Given our proposed algorithms, we implemented them in Hadoop and analyzed their performance characteristics on real datasets (synthetically increased in size).

Our recommendations for implementing each stage are as follows: For the first stage, where data statistics are collected, we recommend the approach that uses two MapReduce phases, one for collecting the statistics and one for aggregating them. For the second stage, where RID pairs of similar records are generated, we recommend the approach that uses one of the existing single-machine set-similarity join algorithms in parallel. For the third stage, where the record pairs are joined and the final result is produced, we recommend the approach that uses two MapReduce phases, one for filling in the first half of each pair and the second for bringing together the two halves.

Chapter 4

Improving Set-Similarity Join Performance Using Adaptive MapReduce

4.1 Introduction

From our study of set-similarity joins in MapReduce in the previous chapter, we observed various limitations of the MapReduce framework. Some of the shortcomings come from the simple execution model offered by the Hadoop implementation of MapReduce, which leads to a limited number of execution strategies. Such limited choices can adversely affect the performance of set-similarity joins. In this chapter we want to improve the performance of set-similarity joins by addressing some of the MapReduce shortcomings.

In this chapter we identify and address three such shortcomings. (1) In the MapReduce framework, there is a tight connection between the number of mappers and

the amount of data that needs to be processed. That is, the input data is divided into equal-sized partitions, called *splits*, and each split is processed by one mapper. Breaking this tight connection by having mappers process more than one split can eliminate some of the scheduling and startup overheads. (2) Before combiners are applied, the (key, value) pairs are grouped by "key" using sort and, for large amounts of data, external memory might be used for sorting. Using a hash table to group the pairs instead could improve the performance, especially if this can be done without using external memory. (3) After a job is started, no changes can be made in the way data is being partitioned over the network. However, by collecting some of the map output, we could possibly decide a better way to partition the data than what we could have decided before the job is started. Moreover, samples collected from map output can have various other uses, such as tuning the hash table used for applying combiners. We will see that such enhancements to the framework can help increase the performance not only for fuzzy-join queries, but also for other common types of queries such as aggregation, join, and sort queries.

New MapReduce-inspired massive data processing platforms, such as Dryad [39], Hyracks [14], Spark [69], and Nephele [9] have also considered these shortcomings in order to achieve more flexibility. They all include elements of MapReduce, but have more choices in runtime query execution. In contrast to these projects, we choose, in this chapter, to enhance MapReduce to leverage existing investments in the Hadoop framework and in the query processing systems built on top of it, such as Jaql [40], Pig [29], and Hive [59].

While adding new runtime options to Hadoop, we want to avoid making performance tuning harder than what it already is [7]. For that reason we use "*adaptive*" algorithms, i.e., algorithms that require very little tuning and can automatically turn certain features on or off based on the input data and the available resources. Adap-

tive algorithms can provide superior performance stability since they are robust to tuning errors and changing runtime conditions such as other jobs running on the same cluster. For example, our MapReduce-set-similarity-join experiments in the previous chapter needed fairly extensive tuning through trial-and-error and log analysis to achieve the best performance. Typical MapReduce tuning includes choosing the split size, input and output buffer size, etc. In such an environment, the use of adaptive run-time algorithms is preferable.

In this chapter, we address the three MapReduce shortcomings outlined above. We propose three adaptive techniques: (1) Adaptive Mappers (AM), (2) Adaptive Combiners (AC), and (3) Adaptive Samplers and Partitioners (ASP). All these techniques are applied in the mappers with the help of an asynchronous communication channel. We call such mappers "situation-aware mappers" (SAMs). The required asynchronous communication is achieved using a transactional, distributed meta-data store (DMDS).

The main contributions of this chapter are:

- We propose SAMs that use an asynchronous communication channel to exchange information about their state and collaboratively make optimization decisions.
- We employ SAMs to build a number of adaptive optimization techniques that preserve the fault-tolerance, scalability, and programmability of MapReduce.
- We demonstrate up to $\times 3$ performance improvements over existing state-of-the-art techniques, as well as excellent performance stability, through an experimental evaluation of our adaptive techniques.

The rest of the chapter is organized as follows. We give an overview of our adaptive

techniques in Section 4.2. We describe the adaptive techniques in detail in Section 4.3. Section 4.4 contains experimental evaluation of all the proposed techniques. We conclude in Section 4.5.

4.2 Overview

In this section we give an overview of the components which enable our adaptive techniques: Situation-Aware Mappers (SAM) and Distributed Meta-data Store (DMDS).

Situation-Aware Mappers: The core idea proposed in this chapter is that of a Situation-Aware Mappers (SAM). The basis for SAM is that it is possible to make MapReduce more flexible and adaptive by removing a key assumption of the programming model namely, the assumption that mappers are completely *independent*. We introduce an asynchronous communication channel between mappers that enables mappers to post some metadata about their state and to see states of other mappers. SAMs can get an aggregate view of the job state to make globally coordinated optimization decisions. In particular, our SAM tasks are able to alter their execution at runtime depending on the global state of the job. However, while implementing SAMs, we will have to be careful not to violate key MapReduce assumptions about scalability and fault-tolerance, and not to introduce noticeable performance overhead.

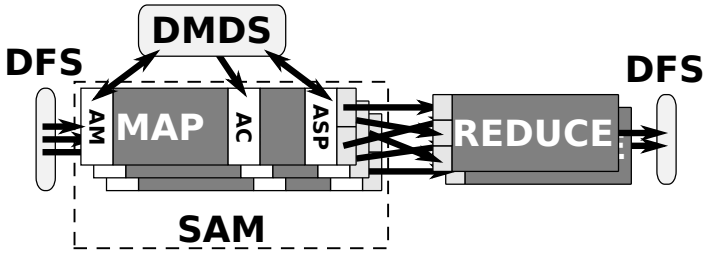


Figure 4.1: Adaptive techniques in SAMs and their communication via DMDS.

Figure 4.1 shows an overview of SAMs and their communication via the DMDS. As

shown, SAMs are composed of Adaptive Mappers (AM), Adaptive Combiners (AC), and Adaptive Sampling and Partitioning (ASP). AMs and ASPs post and receive data using the DMDS, while ACs only receive data. These components will be described in full detail in Section 4.3.

The advantages of our SAM-based techniques are two-fold. First, they make it possible to dynamically alter job execution based on the map input, output, and the environment. This allows us to relieve the user, or the high-level-language compiler, of making the right choices before each job starts. Second, they decentralize the decision-making for such changes by having SAMs collectively make decisions. This reduces the load on the coordinator, preventing it from becoming a bottleneck. It also makes the decision process more flexible, as decisions that affect only the local scope can be made individually by each SAM.

Hadoop's flexible programming environment allowed us to implement SAMs and use them in a variety of adaptive techniques without requiring any changes to Hadoop itself. Instead, the adaptive techniques are packaged as a library that can be used by Hadoop programmers through a simple API. Notice that the original programming API of MapReduce remains completely unchanged, which is obviously very important. In order to make the adaptive techniques completely transparent to the user, we implemented them inside the Jaql [40] query processor, which is an open-source query processing system that compiles its queries into Hadoop jobs.

Our adaptive techniques have been implemented in the context of Hadoop and Jaql, and the rest of the chapter describes them in this context. However, the general ideas of SAMs and adaptive techniques are applicable more broadly. It should be relatively easy to adopt these ideas to other systems that include elements of MapReduce, such as Dryad [39], Hyracks [14], Spark [69], and Nephele [9], not to mention other Hadoop-based query processing systems such as Hive and Pig.

Distributed Meta-data Store: One of the main components of our SAM-based techniques is a distributed meta-data store (DMDS). The store has to perform efficient distributed reads and writes of small amounts of data in a transactional manner. We use Apache ZooKeeper [5, 37], an open-source distributed coordination service, for this purpose. The ZooKeeper service is highly available, if configured with three or more servers, and is fault-tolerant. ZooKeeper's data is organized in a hierarchical structure similar to a file system, except that each node can contain both data and sub-nodes. A node's content is a sequence of bytes and has a version number attached to it. A ZooKeeper server keeps the entire structure and the associated data cached in memory. Reads are extremely fast, but writes are slightly slower because the data needs to be serialized to disk and agreed upon by the majority of the servers. Transactions are supported by versioning the data. Zookeeper provides a basic set of primitives, such as `create`, `delete`, `exists`, `get`, and `set`, which can be easily used to build more complex services such as synchronization and leader election. In case a server fails, clients can connect to any of the remaining servers while sequential consistency is preserved, that is, the clients receive the updates from the server in the same order as if no failure had occurred. Moreover, clients can set watches on certain ZooKeeper nodes and get a notification if there are any changes to those nodes. For the rest of the chapter we refer to DMDS as ZooKeeper, though other transactional stores, such as Memcached or even an RDBMS, could also be used in place of ZooKeeper.

4.3 Techniques to Make MapReduce Adaptive

In this section we describe the details of out techniques for making the MapReduce framework more adaptive to the input data and runtime conditions. These techniques

affect different parts of a MapReduce job, yet all of them are implemented inside of the map tasks and they all rely on asynchronous communication.

4.3.1 Technique 1: Adaptive Mappers (AM)

When a job starts, the MapReduce framework logically divides the input data into equal-sized partitions called *splits*. Each split is then processed by one map instance. As a consequence, the number of map instances is equal to the number of input splits and vice versa. To balance the workload across the cluster, a job can have many *waves* of mappers, i.e., there may be more map tasks than there are map *slots* available to execute them. Having more mappers increases work required for scheduling and starting tasks. The startup overhead may include running user code to perform job-specific setup tasks, such as loading reference data, which can become a significant portion of the running time. At the same time, having smaller splits also tends to reduce the benefit from applying a combiner.

Out proposed AMs preserve the workload balance achieved by having small splits yet eliminates the scheduling and startup overhead introduced by small splits. AMs achieve this by separating the number of map instances from the number of splits. Figure 4.2 shows the difference between regular mappers and AMs. The input data is divided into eight splits, so eight regular mappers are traditionally needed to process all the splits. AMs do not have this constraint. We could start, for example, only four AMs, which could then process two splits each. The exact split assignment is decided at runtime. This is achieved by allowing a given mapper to process multiple splits sequentially.

The separation of the number of mappers from the number of splits is achieved as follows: The split location information is stored in DMDS. A fixed number of mappers

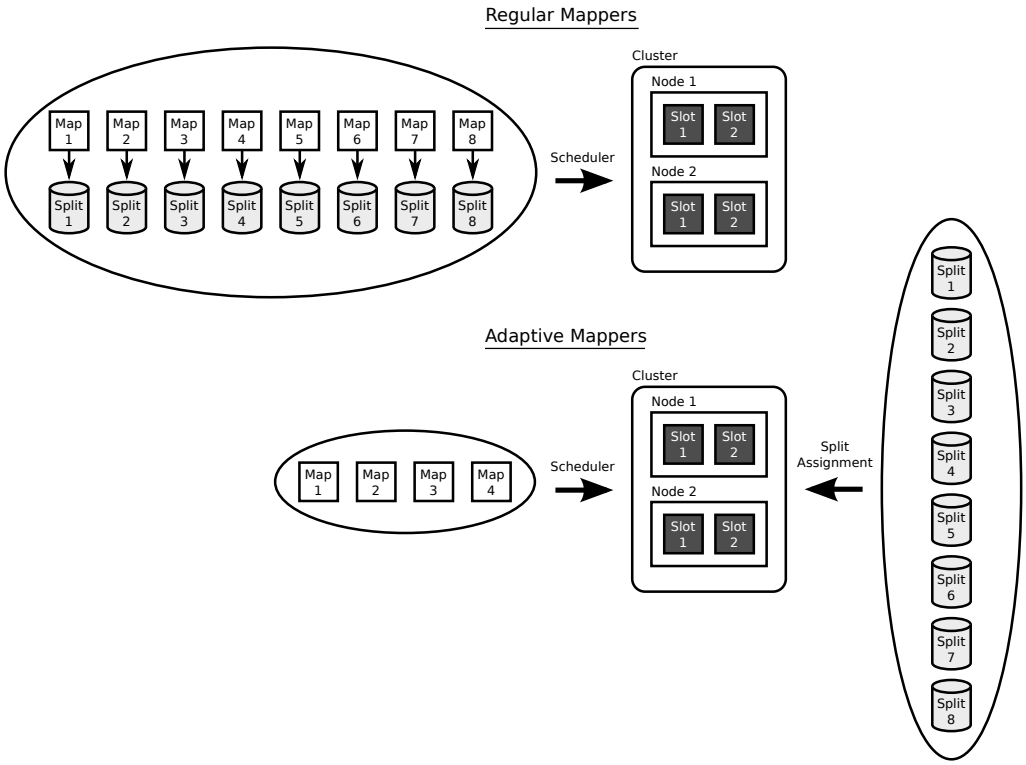


Figure 4.2: Comparison between regular mappers and adaptive mappers (AMs).

are started to compete for splits. Every time an AM finishes processing a split, it makes a decision to stop or to take a new split from DMDS, and to "concatenate" it to the existing one, transparently to the map function. Split assignment conflicts are avoided using the transaction capabilities of DMDS.

Figure 4.3 shows the main flow of execution and the data structures of our ZooKeeper-based AM implementation. In step 1, the MapReduce client creates the AM data structure in ZooKeeper. For each job, we create a `locations` and an `assigned` node. The `locations` node contains metadata for all input splits organized by hosts where these splits are available. On multi-rack clusters, hosts are further organized by racks and data centers. The `assigned` node will contain information about which split got processed by which mapper.

In step 2, the MapReduce client uses *virtual* splits to start mappers. Virtual splits

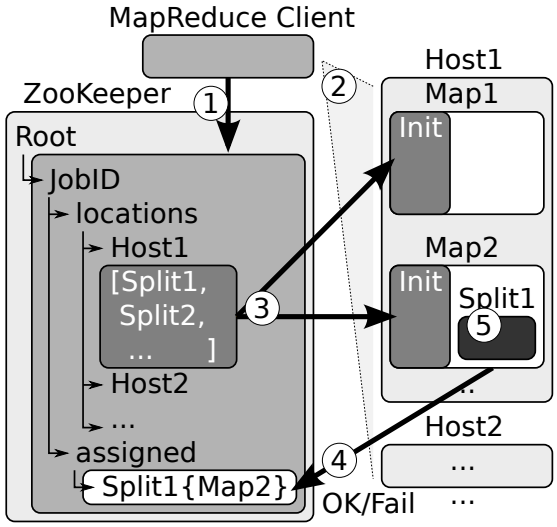


Figure 4.3: Local split assignment in AMs.

look like regular splits to the `map` functions, but then have no actual data attached to them. When the virtual splits are initialized, they connect to ZooKeeper and retrieve a list of the real splits that are local to the current host (step 3). In step 4, the AM picks a random split from the list and tries to lock it. The randomness helps to minimize the number of locking collisions between AMs working on the same host. For example, in the figure, `Map2` tries to lock `Split1` for processing by creating a node for `Split1` under the `assigned` node. The created node contains the ID of the map. If the node creation succeeds the mapper has locked the split and can process it (step 5). After all of the data from the chosen split has been processed, the map will try to pick a new split. If it succeeds, mapper processing continues. The split switch is transparent to the `map` function unless it explicitly asks for the location of the split. Once mappers have finished processing local splits, they start processing any unprocessed remote splits. They do this by selecting a random unfinished host. From the list of splits available at that host and they try to process all the splits not already marked as processed in the `assigned` list. When no more splits are available in ZooKeeper, the mappers end their execution.

Let us see how AMs can be used in fuzzy-join processing. Consider, for example, the

first stage in the set-similarity join algorithm, where we compute the token order. Assume we use the Basic Token Ordering (BTO) alternative for this stage. For the first MapReduce job, `Split1`, `Split2`, etc. in Figure 4.3 correspond to input records. Mappers pick a split using ZooKeeper and, for each record, they tokenize the join-attribute value, outputting one `(token, 1)` pair per token. When the current split has been processed, they get another split from ZooKeeper and continue the processing.

As mappers process splits and accumulate output data, we have to keep in mind that the output data has to be sorted and shuffled, and in case of mapper failure, the data has to be reprocessed. After every split, AMs decide if they should pick another split or stop execution based on how long they have been running, how many splits they have processed, and how much output they have produced. For instance, an AM stops execution if the size of its output exceeds a pre-determined threshold. The intuition is that if the map output size is significant, it will be advantageous to start the shuffle early and overlap it with the mappers. Once a given AM stops, the reducers can start copying its output.

Additionally, a user may choose to specify a time limit t to mitigate the performance impact of failures. An AM will not take any more splits if it has been running longer than t. Given an expected mean time between failures (MTBF), it is possible to automatically optimize t by doing a simple cost-benefit analysis. The cost of taking an additional split is paid for during failures when more work must be redone, but the benefit is the elimination of per-split start-up time during normal processing. Every AM observes how long it has been running and remembers how long its map start-up processing took.

Note that if mappers in the earlier waves decide to stop, mappers in the later stages get to process the rest of the data. The AMs of the last wave do not stop until all

splits have been processed.

Hadoop Implementation Details

We implemented adaptive mappers in Hadoop by creating a new input format and a new record reader. These components wrap the job's original input format and record reader, respectively. When Hadoop initially asks the input format for the splits, the real splits are stored in ZooKeeper and a number of virtual splits are created and returned. When Hadoop asks the record reader to read a record from a virtual split, a real split is fetched and the record is read from it instead.

To facilitate multiple waves of mappers, we start, by default, four times more AMs than the number of map slots in the cluster. From our experience during the experimental evaluation, four waves of mappers provided most of the benefit for overlapping the map and shuffle phases, without introducing too much overhead. If users specify an AM time limit, they should also manually adjust the number of waves based on their expectation of the map running time.

Fault-tolerance

A very important aspect of the MapReduce framework is its fault-tolerance. Because AMs manage the splits themselves, special care needs to be taken to handle task failures. Essentially, as a mapper takes splits, those splits will not be processed by other mappers. If a mapper fails, we need to make sure that all of the splits that it tried to process are eventually processed (possibly by other mappers).

AM failure resolution relies on the fact that MapReduce automatically restarts failed mappers. A restarted AM scans the `assigned` node and removes all entries assigned

to the virtual split of this AM that were locked by the previous execution attempt of the same task. Thus, the splits that were assigned to the previous attempt become available for reassignment. In order for the other mappers to learn about the newly available splits, they read the updated global list of assigned splits (in the `assigned` node) when they run out of splits. In this way, other mappers can share the workload that needs to be redone due to the failure.

Scheduling Support

AMs take special care to cooperate with the MapReduce scheduling algorithms. The default Hadoop scheduler is based on a FIFO queue and it always assigns all of the map tasks of a given job to the available slots before taking any map tasks of the next job in the queue. The FIFO scheduler operates the same way with both regular mappers and with AMs.

In contrast, the FAIR scheduler [68], which has gained popularity in large shared clusters, divides slots between multiple jobs and schedules tasks of all the jobs at the same time. FAIR avoids the potential temporary starvation of a smaller job that arrives in the system after a large job by reducing the number of slots allocated to the large job. This is because, when a map task of a large job finishes, its slot can be used to run the tasks of a smaller job. Thus, the large job gets throttled back to let the small job finish first. The FAIR policy typically results in better average response times for heterogeneous batches of jobs than FIFO. Notice that FAIR relies on large jobs having many waves of mappers to make it possible to throttle them down. This is usually the case for normal mappers. However, recall that AMs may finish an entire job in one wave. In that case, FAIR performance will become equivalent to that of FIFO.

To support FAIR scheduling, AMs include a mechanism to respect slot allocations and to shut down some mappers if the allocation for that job is reduced. To achieve this, we store the number of slots allocated to every job in ZooKeeper. We introduced a small modification to the FAIR scheduler code so that every time this number gets changed by the scheduler it is also updated in ZooKeeper. We also maintain the number of currently running AM tasks for each job in ZooKeeper. Every time a non-last-wave AM tries to take a new split, it reads these two counters for its job. If the number of running AMs exceeds the current allocation, the AM terminates. The last wave of AMs do not do this check, however, in order to guarantee job completion. Note that if FAIR increases the slot allocation for an adaptive job, new AMs will be started by Hadoop.

For more advanced schedulers that need to understand the performance characteristics of a job, other changes may be needed to support AMs. For example the FLEX scheduler [61] would also have to be updated to read from ZooKeeper in order to understand how much progress a job has made.

4.3.2 Technique 2: Adaptive Combiners (AC)

MapReduce supports local aggregation of map outputs using *combiner* functions to reduce the amount of data that needs to be shuffled and merged in the reducers. Hadoop combiners require all map outputs to be serialized, sorted, and possibly written to disk. However, hash-based aggregation could perform better than sort-based aggregation, especially when grouping is performed on relatively low-cardinality attributes, as compression is achieved by the aggregation.

In this section we describe how we leverage hash-based aggregation for combining map outputs with keys that appear many times, while keeping sort-based aggregation as

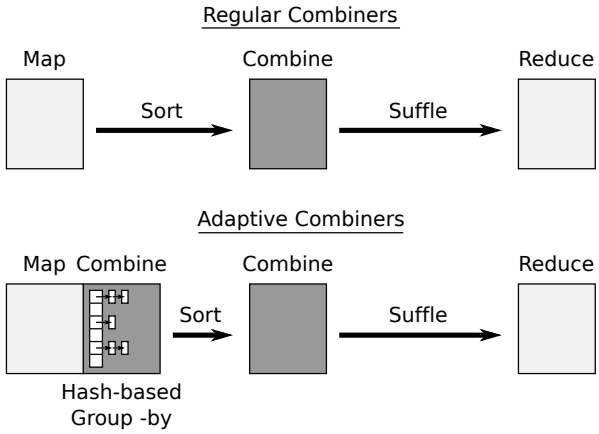


Figure 4.4: Comparison between regular combiners and adaptive combiners (ACs).

a fallback alternative for non-frequent keys. Figure 4.4 shows the difference between regular combiners and Adaptive Combiners (ACs). ACs use a hash table to group the (key, value) pairs by key in the mapper and applies the combiner for each group. Regular combiners are still used as a fallback mechanism to do additional aggregations not done by ACs, due to insufficient memory.

ACs preserve the benefit of shuffling and merging less data in the reducers while eliminating some of the overhead required to apply combiners. Out AC mechanism replaces MapReduce's sort-based aggregation with hash-based aggregation for frequently occurring map output keys by maintaining a fixed size cache of partial aggregates (implemented as a hash-map). For each map output R, the cache is probed with R's key. On a cache hit, the combine function is applied for the output value and the cached value, and the result is stored back into the cache. On a cache miss, if the cache is not full, we create a new entry for the output pair. If the cache has reached its size limit, one of its entries has to be output. Depending on the cache replacement policy, we either directly output the current entry (*No-Replacement* policy) or we insert the current `(key, value)` pair into the cache and then remove and output the least-recently-used (LRU) entry from the cache (*LRU* policy). The No-Replacement (NR) policy assumes that the most frequent keys will be inserted into the cache before

it gets full. If the key distribution is uniform or normal, and in no particular order, than on average the first set of keys that will fit into the cache are as good as any set of keys. This cache policy has a very small overhead as no deletions are performed. In LRU we insert the current pair in the cache and then remove and output the pair with the least-recently-used key from the cache. The main idea of this policy is to keep the most popular keys in the cache in order to maximize the aggregation opportunities. For instance, LRU is an optimal policy if data is pre-sorted on the output key, whereas NR may perform very badly in this case, depending on the order. Other policies could also be implemented. Finally, when there is no more input for the map, we scan the cache and write all the pairs to the output.

Continuing our fuzzy-join stage 1 example from Section 4.3.1, when ACs are used, the mappers do not output the `(token, 1)` pairs directly (see Figure 4.4). For each pair we probe the AC cache using the token as the key. If the key exists, we add one to the current value. If the key does not exist and the cache is not full, we create a new entry for the key with the value of 1. In both cases the produced pair is then discarded as it has been accounted for in the cache. If the cache is full and we are using the NR cache replacement policy, the pair is written to the output. For the LRU cache replacement policy, the pair is inserted into the cache and the least-recently-used key from the cache is removed and written to the output.

Note that ACs are best-effort aggregators. That is, they might not perform all possible aggregations, but they will never spill to disk. In fact, the regular combiners are still enabled, and they will be able to perform any additional aggregations that were not done by the cache. It is also worth noting that the ACs operate on deserialized records and reduce the amount of data that regular combiners have to serialize and sort. Moreover, notice that the ACs benefit increase when they are used with AMs, as multiple splits can be processed by an AC without draining and rebuilding the

cache.

Multi-core Optimizations: To make use of multi-core machines available to a Hadoop cluster, usually multiple mappers are scheduled to run in parallel in different processes, on a single node. Using ACs, each mapper will have its own cache. One such cache can only use a fraction of the memory and do a faction of the possible aggregations. One way to overcome this is to have a single map process with multiple threads and a single shared cache. ACs could be easily adapted to work in a multi-threaded setting using ideas that have been explored for hardware caching of hash tables on multi-core machines as in [19].

4.3.3 Technique 3: Adaptive Sampling and Partitioning (ASP)

The partitioning function of a MapReduce job decides what data goes to which reducer. In Hadoop, by default, the partitioning is done by hashing, though a custom partitioning function may be used (e.g., for range partitioning for global sorting, or for performance reasons). Regardless of whether the partitioner is custom or not, the partitioning function is statically decided before the job starts. In cases when good partitioning depends on the input data, a separate sampling job is often used. However, such sampling can be expensive as it is not clear how much input data the mappers need to process to produce sufficient output between all of them. Also, such sampling effort is essentially wasted when all of the data are reprocessed by the main job.

In contrast, we propose an Adaptive Sampling (AS) technique that collects a sample of map output keys and aggregates them into a local histograms. The local histograms of all the mappers are accumulated by one of the mappers whose job is to compute a global histogram. Our Adaptive Partitioning (AP) technique then uses the global

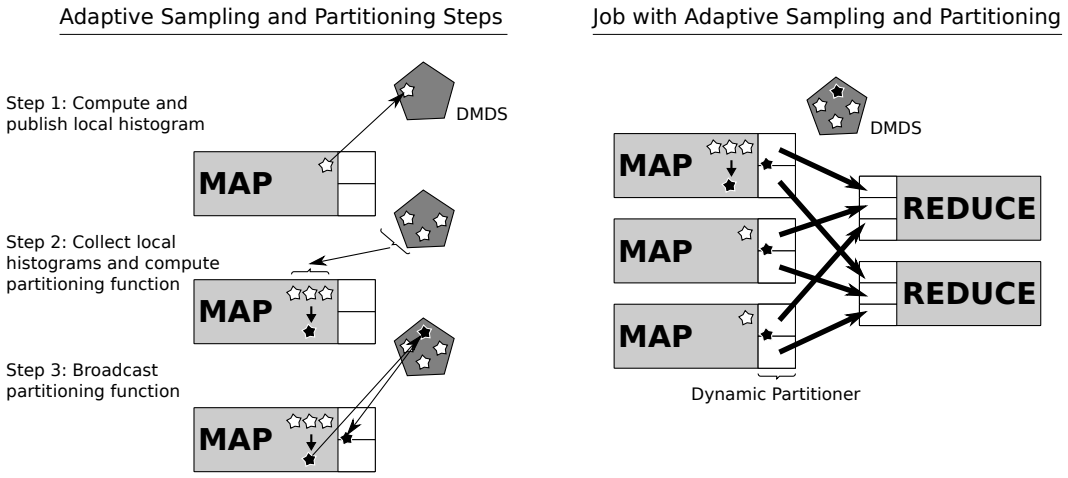


Figure 4.5: Overview of adaptive sampling and partitioning (ASP).

histogram produced by AS to dynamically tune the partitioning function while the job is running. That is, AP piggybacks on the mapper that computes the global histogram and, based on the histogram, computes the partitioning function. The partitioning function is then broadcast to all the partitioners using the DMDS. Figure 4.5 shows an overview of the two techniques.

AS eliminates the need for a sampling stage in which work is wasted, balances the sampling effort across cluster, and avoids sampling either too much or too little. That is, AS dynamically decides when to stop sampling based on a global sampling condition. Besides being used by the AP, the global histogram produced by AS has many other potential applications, for example, setting various parameters for the hash-tables used by AC. (See Section 4.3.3 for a full list.)

Adaptive Sampling (AS)

The main idea of this technique is to have mappers independently sample their output data while coordinating to meet a global sampling requirement. After the global sampling requirement is met, a leader mapper is elected to aggregate the samples.

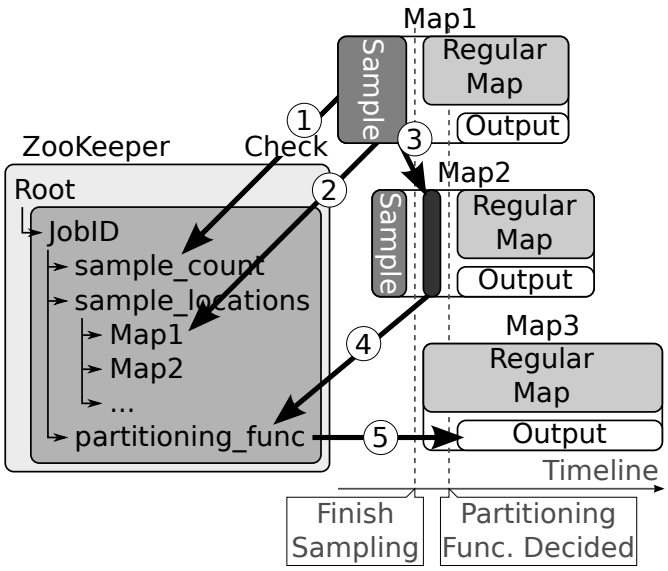


Figure 4.6: Communication between mappers and ZooKeeper in ASP for performing a global sort using a fixed number of samples (`samples_count`).

The coordination between mappers is achieved using ZooKeeper.

AS depends on AMs for two reasons. First, AMs randomly choose input splits to ensure a random block-sampling, that is, records are chosen from the input splits randomly selected for processing by AMs. Second, AMs guarantee that enough map outputs are generated in the first wave by not stopping execution until the histogram is produced.

AS has two phases, a Sample-Collection phase and a Sample-Aggregation phase. In Sample-Collection, the status of the global sampling requirement is stored in a predetermined place in ZooKeeper. An example of a global sampling requirement is a fixed number of samples. In this case, ZooKeeper stores the current sample count. Figure 4.6 shows the communication between the mappers and ZooKeeper during AS (steps 1 to 3). When mappers start, they check ZooKeeper to see if the global sampling requirement is met (step 1 in the figure). If the requirement is not met, the mappers start sampling their outputs. Once a mapper accumulates a small fraction of the required samples (e.g., in the case the fixed number of samples, a good fraction

is 1%), it updates the sample count in ZooKeeper, writes the sample to the local disks, and publishes the host and file-name in ZooKeeper (step 2 in the figure). Once a mapper observes that the global requirement is met, it stops sampling and applies to be a leader by election, again using ZooKeeper. Once a lead mapper is elected, the leader queries ZooKeeper for the sample locations; it then retrieves and aggregates all the samples (step 3 in the figure). The overhead introduced by the leader election and the sample aggregation is small (1-2 seconds). The non-leader mappers continue their regular map processing.

Depending on the reason why sampling is necessary, the mappers could directly output the processed data or they might have to buffer it until the sampling process has finished. If the sampled data is necessary for partitioning, the mappers cannot output any data until the leader aggregates the samples and the partitioning map is computed (see the AP description below). In this case, the mappers each allocate a buffer for storing processed data until it can be output. In case the output buffer becomes full, the mappers have the option of stalling, writing the processed data to disk, or just discarding and reprocessing the data after the sampling is completed. In the case where the sampled data is necessary for optimizations in the reduce phase or in the following jobs, the mappers can output the processed data directly without having the need to buffer it.

Consider for example a multiple-input multiple-output global sort job, i.e., when the input and output data are on multiple nodes and the output data is spread across the nodes in sorted order. We need to choose a range-partitioning map for partitioning the map outputs so that the outputs will be evenly distributed across reducers and are in an increasing order. We use AS to sample the map output keys in order to determine the range partitioning map. Suppose we decide to collect 5,000 samples. Initially, each mapper buffers the output pairs and writes them in the sample file without

outputting any pairs to the regular output stream (as the partitioning function has not been decided yet). After every 50 samples are collected (1% of 5,000), the mappers increment the global sample count in ZooKeeper. The mappers also check the current value of the counter, and when they detect that the counter has reached 5,000, they stop and do a leader election. The elected leader mapper collects all the samples.

Note that the sampling process is balanced across the cluster due to the global coordination between the mappers. Mappers might end up sampling different amounts of data depending on when they are scheduled. Early mappers will sample more, while later mappers might not sample at all.

Implementation Details: The status of the global sampling requirements are stored as the data of a predetermined node in ZooKeeper. After computing a sample fraction, each mapper reads the current state of the global requirement, updates it locally, and writes it back to ZooKeeper. Note that for performance reasons we do not perform locking of the ZooKeeper node, so state updates might be overwritten and more than the required number of samples might be collected. This is not a problem if the samples are easy to produce. In the case where samples are expensive to produce, the chance of multiple mappers updating the ZooKeeper node in the same time decreases, and so the chance of producing more samples than required decreases. For implementing the leader election we use ZooKeeper Sequential nodes as suggested in the ZooKeeper documentation [5]. Sampled data is transferred between nodes using the same mechanism as for transferring data between mappers and reducers.

Various global sampling requirements can be easily implemented. Some of them could include obtaining a certain coverage of the space. For example, if a certain range has high frequency variations, more samples could be requested in that range if some information about input data partitioning is available.

Fault Tolerance: ZooKeeper contains sufficient information about the state of the Adaptive Samplers so that mappers can recover their state in case of restarts. Still, because of the global coordination requirements, the failure of a mapper could slow down other mappers. In the following, we describe two changes to the coordination algorithm to improve recovery for the most probable types of failures, that is, failure of a node that samples. If there is a software failure, the samples already computed are still accessible to the leader and need not be recomputed. If there is a hardware failure, the sampled data is lost, and recomputing it might be time consuming. To avoid this inefficiency, we change the sampling process so that mappers still keep sampling after the global sampling requirement has been met, doing so until the leader mapper finishes aggregating the samples. In this way, the lost set of samples can be replaced with a new set computed by other mappers in parallel. A less probable type of failure is leader failure. To deal with this situation, we change the leader election process so that non-leader mappers watch for leader failures using ZooKeeper's node watching mechanism. If such a failure is detected, one of the non-leader mappers becomes the new leader and the sample aggregation process is restarted.

Adaptive Partitioning (AP)

AP determines the partitioning function while the job is evaluated. The main idea is for mappers to start processing data, but not produce any output. In parallel, mappers coordinate and the partitioning function is decided by one of them based on the data seen so far. As soon as the partitioning function is decided, the mappers can start outputting data.

The AP piggybacks on AS, which has already aggregated the seen map outputs into a single histogram at a leader mapper. Based on this histogram, the same leader computes the partitioning function and publishes it in ZooKeeper (step 4 in Figure 4.6).

For example, if range partitioning is needed to perform a global sort, AP will split the histogram into contiguous key ranges with approximately the same number of total occurrences. As soon as the partitioning function becomes available in ZooKeeper, the mappers start outputting data, which triggers the start of their partitioners. Each partitioner, upon start-up, loads the partitioning map from ZooKeeper (step 5) and the job continues normally.

Continuing our global-sort example in Section 4.3.3, we can use AP to determine the range-partitioning map. The AP processing happens at the leader mapper where all the samples are collected. Suppose the reduce phase is going to run on k reducers. Then, the samples are sorted and partitioned in a sorted order into k buckets. The partitioning-range-value split points form the partitioning map. The leader mapper stores the partitioning map into ZooKeeper. As the non-leader mappers detect the presence of the partitioning map in ZooKeeper, they first output all the map output pairs that they have been buffering and then continue their normal execution. As soon as the map outputs are produced, the Hadoop framework initializes the partitioners. The initialization code in the partitioner fetches the partitioning map from ZooKeeper and uses it via the partitioning function to partition the map outputs. After the partitioning is completed, the map output pairs are range-partitioned and balanced at the reducers.

Uses of Adaptive Sampling and Partitioning

In this section, we look briefly at on how Adaptive Sampling and Adaptive Partitioning could be used as primitives to obtain more optimization opportunities for MapReduce jobs.

Global Sorting: To perform a global sort, Adaptive Sampling can be used, similar

to [27], to sample the data and Adaptive Partitioning can be then used to decide the range partitioning points to balance the data across the cluster.

Joins: Adaptive Sampling and Partitioning can be used to perform the following optimization in the case of a redistribution equi-join: In this MapReduce join algorithm, provenance labels are added to the each `(key, value)` pair, for example "`R`" for pairs coming from the first dataset and "`S`" for the pairs coming from the second dataset [29]. In reducers, for each unique key, all the "`R`" pairs are read first and buffered in memory, then the "`S`" pairs are streamed by. Using Adaptive Sampling, we can detect which of the two datasets has a smaller set of values for each unique key and then assign the first label, "`R`", to it. In this way, we can reduce the memory utilization in the reducer. This optimization may be especially useful in the case of foreign-key joins. Another possible optimization is to dynamically switch between different join algorithms [13]. Moreover, similar to the global sorting case, Adaptive Sampling and Partitioning can also be used to better balance the workload among reducers in case of skewed joins.

Number of Reducers: Besides balancing the data across the cluster, Adaptive Partitioning could be used to intentionally unbalance the data. One example where this could be useful is when the mappers are very selective and, as a result, the amount of data that need to be processed by the reducers turns out to be very small. In this case, the pre-allocated number of reducers could waste significant time due to startup and shutdown overheads. Instead, we could detect such situations in the Adaptive Partitioner and direct all of the data to a single reducer. The rest of the pre-allocated reducers will then terminate immediately after the mappers end.

Adaptive Combiner Tuning: Adaptive Sampling can also be used for tuning ACs. More exactly, we could exploit the global sampling mechanism to choose the right configuration parameter for the AC cache. That is, by looking at the global

distribution of the data, we could decide (1) whether to use cache or not and (2) what cache policy to use. For example, we could make these decisions based on the following heuristic: First, if the number of distinct keys in the sample is similar in size (e.g., over 75%) to the sample, we should simply disable AC. If the frequency of the top most frequent keys (e.g., top-10) is a significant portion (e.g., over 1%) of the sample size, or we detect that the keys are ordered, we would chose to turn on AC with the LRU replacement policy. Otherwise, AC with NR policy would be used. We can also set the cache size based on the average size of a map output record, which we can measure in AS.

Monitoring: While a job is running, AS could be used to monitor or debug the progress of the mappers. As sample locations get published in ZooKeeper, the UI could periodically fetch the samples and present them to the user for inspection.

4.4 Experiments

In this section we describe a performance evaluation of set-similarity joins using our adaptive techniques. We also evaluate several other common types of queries such as group-by and join. We will examine results for the following types of queries:

(A) *Set-Similarity Join*: We used the same two datasets as in Chapter 3, namely **DBLP** and **CITESEERX**. We scaled up the datasets by $\times 10$ or $\times 100$ as described there, and we stored the datasets in HDFS using text files.

(B) *GROUP-BY*: A GROUP-BY job operates on a single dataset. It groups records by one, two, or three columns and applies an aggregation function for each group. For this task, we used a synthetically generated dataset, called **TWL** (which stands for Transaction WorkLoad), where records contain four integers, including three dimen-

sions (A1, A2, A3) and one fact, similar to a star schema. The dataset had 10 billion records with approximately 12 bytes per record (using variable-length encoding for integers) and a total size of 120GB. We stored the dataset in HDFS using Hadoop Sequence Files. The distributions for the four fields were as follows: A1 and A2 each had a truncated normal distribution with values between 0 and 300 with a mean of 150 and a standard deviation of 37.5. A3 had a truncated normal distribution between 0 and 100 with a mean of 50 and a standard deviation of 12.5.

We used Jaql as a high-level query language to generate the corresponding MapReduce jobs for this type of query. Jaql provides a `batch` statement which groups records in batches. Each batch is read and written as a single record. An `expand` statement is subsequently used to process the records of a batch. Since the records are small, we grouped the input data into 100-record batches to mitigate per-record overhead of HDFS and Sequence Files. The schema, data value distributions, and the queries for this task were based on that of a real IBM customer workload [8].

(C) *JOIN and JOIN-ORDER-BY*: The JOIN and JOIN-ORDER-BY jobs were run over a simulated one-to-many relationship. The goal was to simulate a join between an "`ORDER-ITEM`" table and a "`PRODUCT-SUPPLIER`" table. The "`ORDER-ITEM`" table would contain order items with a "`Product-ID`" column. The "`PRODUCT-SUPPLIER`" table would contain "`Product-ID`", "`Supplier-ID`" pairs that specify which products are provided by which suppliers. The join would be on "`Product-ID`". We used the Sort Benchmark[1] data generator available as part of the Hadoop code-base to generate the actual "`ORDER-ITEM`" table; each record is a sequence of 100 bytes where the first 10 bytes constitute the "`Product-ID`" column. We then simulated the "`PRODUCT-SUPPLIER`" table using a function. That is, we used a function f which takes as input a "`Product-ID`" and returns the number, k, of "`PRODUCT-SUPPLIER`"

[1] http://sortbenchmark.org/

records which would contain that "`Product-ID`". The simulated join is then computed as follows: For each "`ORDER-ITEM`" input record, we apply the function f and then output the input record k times. We generated between 10 million and 1 billion "`ORDER-ITEM`" records for a total size between 0.9GB and 93GB. We refer to this dataset as the **TERASORT** dataset. We stored the dataset in HDFS using a custom binary file format provided by Hadoop for the TERASORT data [52]. Both JOIN and JOIN-ORDER-BY were performed using the broadcast-join algorithm.

Hardware: We ran all of the experiments reported in this chapter on a 42-node IBM System x iDataPlex dx340. Each node had two quad-core Intel Xeon E5540 64-bit 2.83GHz processors, 32GB RAM, and four SATA disks. The cluster thus consisted of 336 cores and 168 disks. The nodes of the cluster were connected using a 1Gbs Ethernet connection. We used one node for running the master daemons (Task-Tracker and Node-Tracker) that managed the Hadoop jobs and the Hadoop distributed file system and 40 nodes for running the slave daemons. On each slave node we allocated four map slots and four reduce slots, so in total we had 160 map slots and 160 reduce slots available.

Software: On each node we installed a Ubuntu Linux operating system with the kernel version 2.6.32-24 64-bit server edition, Java version 1.6 64-bit server edition, Hadoop version 0.20.2, and ZooKeeper version 3.3.1. To maximize parallelism and minimize the running time, we made the following changes to the default Hadoop configuration: We used a 512MB sort buffer with 25% allocated for book-keeping information in the mappers, a 200MB merge buffer in the reducers, a merge factor of 300 in the mappers and the reducers, a 128MB DFS block size, a 128KB I/O buffer size, and a replication factor of one. We disabled speculative execution, started reducers when mappers start, reused the JVM, and used a 4GB JVM heap space. We started three ZooKeeper servers, each on a different node. We ran each experiment

three times and we report the average running time.

4.4.1 Performance of Adaptive Mappers

First, we evaluated the overhead introduced by AMs and ZooKeeper. To properly isolate these overheads, we used a map-only job which processed a fixed-size. The input was divided in different split sizes. The total compute time inside the mappers was fixed. We ran the job using regular mappers and AMs. This experiment was intended to study (1) the ZooKeeper overhead with a varying number of splits and (2) the Hadoop starting and scheduling overhead with a varying number of regular mappers.

To fully isolate the ZooKeeper and Hadoop overheads and do minimum I/O, we used an unrealistically small dataset of just 2000 records, each of only 1 byte. Moreover, to stress ZooKeeper, we used an unrealistic processing time, of 1 second per "record". We used a 5-node cluster with 20 map slots, so ideally the job would always take 100 seconds to finish. Typical jobs would take more than that. We divided the input data of the job into 20, 200, or 2000 splits. The regular mappers job created as many `map` tasks as there were splits. On the other hand, the AMs processed all of the splits using a fixed number of mappers. Thus, regular mappers thus needed 20, 200, or 2000 mappers, respectively. With AMs we always used 20 mappers. Since we ran the job with 20 map slots, the regular-mappers job needed 1, 10, or 100 waves of mappers, respectively, while the AM job always used a single wave. (However, each AM needed to process on average 1, 10, and 100 splits, respectively.) Due to the 1-second-per-record processing time, processing a split would take 100, 10, 1 seconds, respectively.

Figure 4.7 shows the total running time using both regular mappers and AMs, while

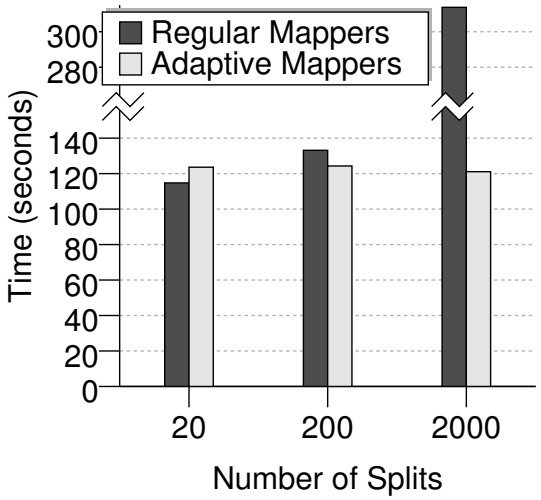


Figure 4.7: Comparison between the performance of regular mappers and adaptive mappers for a map-only sleep job on a 20-slot cluster.

varying the number of splits. The time difference between the regular-mapper tests for different number of splits is due to the overhead incurred by Hadoop for communication, scheduling, and starting map tasks. AMs do not have this overhead. (Recall that JVMs were reused across tasks.) As we increase the number of splits, the AMs' overhead actually decreased because the AMs had more splits to chose from and the probability of a collision when locking a split decreased (see Section 4.3.1 for locking details). From the system logs we saw that on average it took around 10ms to lock a split once an AM was initialized.

Next, we evaluated the performance of AMs on the third stage of our MapReduce set-similarity-join implementation using the One-Phase Record Join (OPRJ) alternative. This algorithm uses one MapReduce job that broadcasts the list of joining RID-pairs to every mapper and streams the original input data to produce the complete join results. We computed the join between the DBLP and CITESEERX datasets scaled up $\times 10$ on the title and author columns, using Jaccard similarity and a 0.90 threshold.

Figure 4.8 shows the running time of OPRJ using regular mappers with different split

sizes and adaptive mappers. For this experiment we used only 5 cluster nodes (with 40 map slots) to ensure that every node got a large amount of data. For regular mappers, the 2048MB split size generated a single wave of mappers. At each of the following steps we reduced the split size by half and, as a consequence, the number of map waves doubled. AMs used a single wave of mappers. The poor performance of the regular mappers as the split size decreased was due to two overheads: (1) The time needed to load (repeatedly) the list of RID-pairs in memory each time a mapper starts, and (2) the Hadoop overhead for scheduling and starting the map tasks. Even when we used a 2048MB split size, regular mappers were still slower than AMs because AMs balanced the workload better across the cluster. We can see that AMs improved the performance by three times compared to the default Hadoop split size 64MB), 166 seconds for AMs versus 583 seconds for regular mappers.

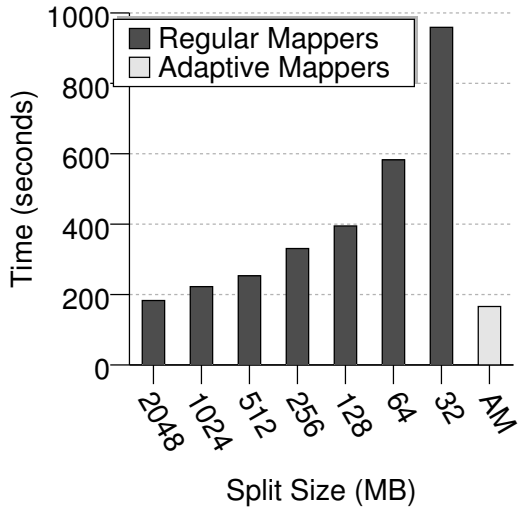


Figure 4.8: Performance comparison between regular mappers and adaptive mappers on the third stage of Set-Similarity Join, One-Phase Record Join of DBLP ×10 and CITESEERX ×10.

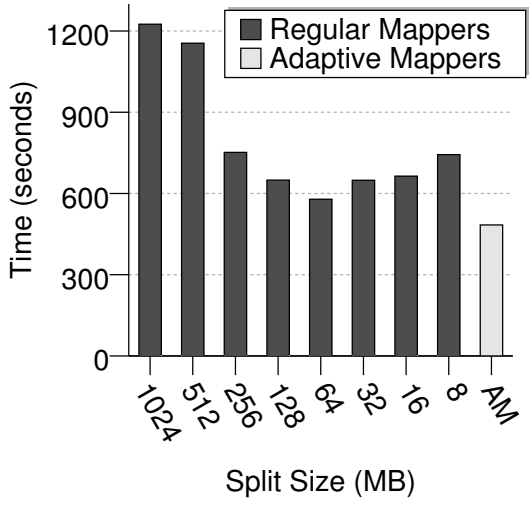


Figure 4.9: Performance comparison between regular mappers and adaptive mappers on the synthetic JOIN query involving 1 billion TERASORT records.

Last but not least, we ran a map-only job that performed a synthetic broadcast-join

on 1 billion TERASORT records. We assumed that the records were sorted by the join key, which in turn was correlated to join fan-out. Figure 4.9 shows the running time of this job using regular mappers with different split sizes and adaptive mappers. For regular mappers we varied the split size between 8MB and 1024MB. The 1024MB-split case used a single wave of mappers. AMs also used a single wave of mappers. We can see that regular mappers performed the worst when we used the largest block size. This is because some input splits had many keys with a large fan-out, while others had many keys with a very small fan-out. This created an imbalance between the mappers when we used a large split size. As we decreased the split size, the workload balance improved. AMs were even faster than the fastest regular-mapper setting (64MB splits). This is because the regular-mapper setting used 20 waves of mappers, which introduced significant scheduling and startup overhead.

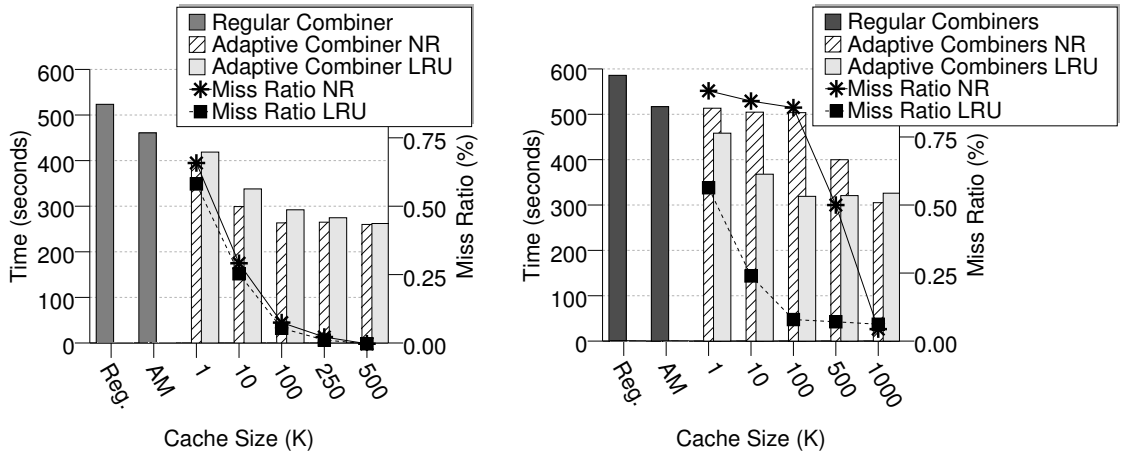
We conclude that AMs can significantly improve the performance and robustness of MapReduce jobs on different workloads. They minimize the `map` startup overhead and balance the workload across the cluster. Moreover, as we will see next, AMs and improve the performance of MapReduce jobs even further when used in conjunction with ACs.

4.4.2 Performance of Adaptive Combiners

We evaluated the performance of AC on two workloads. We focused on the first stage of our MapReduce implementation of set-similarity joins, using the Basic Token Ordering (BTO) alternative (see Chapter 3). This job tokenizes the join attribute values from each dataset, normalizes the tokens, and computes token occurrence frequencies. Since dataset sizes are typically large relative to the dictionary size, the use of combiners to compute partial counts for each token provides a very important

performance boost.

For this experiment, we computed a self-join on the DBLP dataset scaled ×100 to 32GB of text. The number of unique tokens was scaled up to ×10, to around 5 millions, while maintaining the original long tail distribution of occurrences.



(a) DBLP ×100 with ×1 unique tokens.

(b) DBLP ×100 with ×10 unique tokens

Figure 4.10: Performance comparison between regular combiners and adaptive combiners on the first stage of Set-Similarity Join, Basic Token Ordering.

Figure 4.10(a) shows the running time for the first MapReduce job of the BTO algorithm for the following cases: (1) regular mappers and combiners ("Reg."), (2) AMs and regular combiners ("AM"), and (3) AMs and ACs with different cache sizes and cache-replacement policies, including *No-Replacement* (NR) and *Least-Recently-Used* (LRU). The bars represent the running times of the job, while the lines represent the cache miss ratios (indicated on the right-hand-side axis). We first used DBLP ×100 without increasing the number of unique tokens. We varied the cache size from 1K to 500K entries. The total number of unique tokens was around 500K. We can see that even for small cache sizes (1K) and significant miss ratios ($58\% - 65\%$), ACs were faster than regular combiners. For larger cache sizes (100K-500K), ACs provide an almost ×2 improvement over regular combiners. The miss ratio of the two replacement policies were very similar, and we can see that LRU was more expensive than NR

due to the cost of maintaining the least-recently-used list. Figure 4.10(b) shows the same setup but with ×10 more unique tokens. In this experiment we varied the cache size from 1K to 1,000K records. We can see that LRU provided a better performance than NR on small cache sizes. This is due to the very skewed distribution of the tokens, as the frequent tokens are better captured by the LRU policy. However, LRU performance did not improve once the cache size grew above 100K entries, as the LRU cache-maintenance costs offset the gains from the extra cache hits. The performance overhead of NR was negligible even when it had a very poor miss ratio due to small cache sizes. Using AMs and ACs with a cache size of 1,000K we obtained about ×2 improvement over regular MapReduce jobs. Note that given the input data with the largest cache size we could only cache one fifth of the unique tokens.

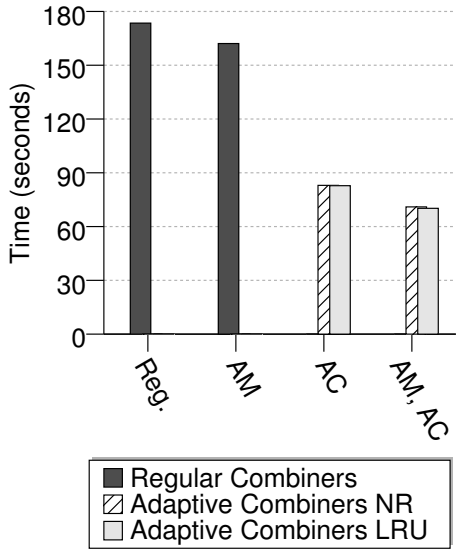
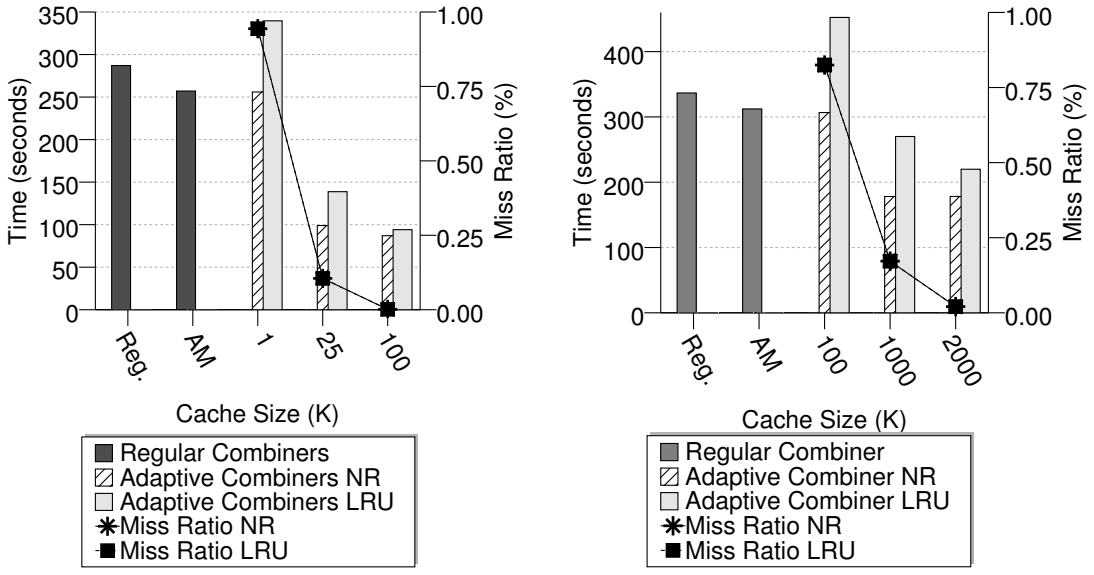


Figure 4.11: Performance comparison between regular combiners and adaptive combiners with various degrees of adaptive mappers on a GROUP-BY query on dimension A1 of the TWL dataset.

The second set of experiments for evaluating the performance of ACs used on the GROUP-BY task. The aggregation function was `count`. Figure 4.11 shows the running time of the query for grouping on the A1 dimension for the same combinations of AMs and ACs as in the previous experiment. For the settings of ACs we used a

cache size of 1,000 entries. The A1 dimension had 300 unique grouping keys, so they all fit in the cache. We can see that with regular mappers, ACs provided a better performance than regular combiners by a factor greater than ×2. Adding AMs did not improve the performance significantly. Since all the keys fit in the cache, we had a 0% miss ratio and there was little to no difference between the two cache policies.



(a) On dimension A1 and A2.

(b) On dimension A1, A2, and A3.

Figure 4.12: Performance comparison between regular combiners and adaptive combiners on a GROUP-BY query on the TWL dataset.

Figure 4.12(a) shows the running time for the same query, but grouping on both dimensions A1 and A2. Over the two dimensions we had 90,000 unique composite grouping keys. The figure also shows the miss ratio for each cache size. We can see that for the largest cache size that could accommodate all the keys, the ACs achieved a ×3 performance improvement over regular combiners. For the small-cache-size case, the LRU policy became very expensive. This is due to the more complex data structures that have to be maintained and the replacements on every miss. All this cost was paid without improving the miss ratio significantly. This cost is relatively high compared to the small cost of applying the combiner. Still, the NR cache policy

performed significantly better for the small cache sizes and is at worst on par with using a regular combiner.

Figure 4.12(b) shows the same GROUP-BY query when the grouping dimensions were A1, A2, and A3. Over the three dimensions we had around seven million unique composite grouping keys. As we can see, ACs achieve an almost $\times 2$ improvement over regular combiners when used with 1,000K cache size and the NR cache policy. LRU exhibited similar behavior as in the previous experiment. An interesting observation is that very large cache sizes (4,000-5,000K entries) actually negatively affected the performance due to the fact that they were expensive to build and occupied considerable resources, while they still did not reduce the miss ratio significantly.

Overall, we conclude that group-by jobs can significantly benefit from using ACs. When used in conjunction with AMs, the performance continues to improve. The main benefit of ACs over regular combiners is that ACs do not have to serialize, sort, and deserialize the data in order to apply the `combine` function. As for AMs, by stitching splits together, AMs allow for the cache to be reused. In general the NR cache policy brings significant improvements, but for cases where the cached keys follow a power-law distribution and we can only afford a small cache size, the LRU policy performs better than NR.

4.4.3 Performance of Adaptive Sampling and Partitioning

In this section we analyze the performance of the AS and AP features using the same broadcast-join query on TERASORT records that was used in Section 4.4.1, but then followed by a global sort of the results. The join is performed in the mappers using a broadcast-join approach.

To produce the final global order, a range-partitioning map has to be computed based on a sample. That is, we collect a set of samples, sort them and then divide them in equal size ranges. The range splitting points form the partitioning map. We tried the following four methods for answering this query:

1. "Client Input Sample" followed by "Join and Sort Job": Before the job is launched, we sequentially sample input data (as in the Terasort package in Hadoop). From the sample, we compute the partitioning map and store it in HDFS. Finally, we run a job that does both the join and sort operations using the partitioning map from HDFS.

2. "Client Join Sample" followed by "Join and Sort Job": Similarly as in (1) we sequentially sample input data before the job is started, but we instantiate and apply the `map` function. We collect the results of the `map` function and we compute a partitioning map, which we then store in HDFS. Finally, we run the join-and-sort job using the partitioning map stored in HDFS.

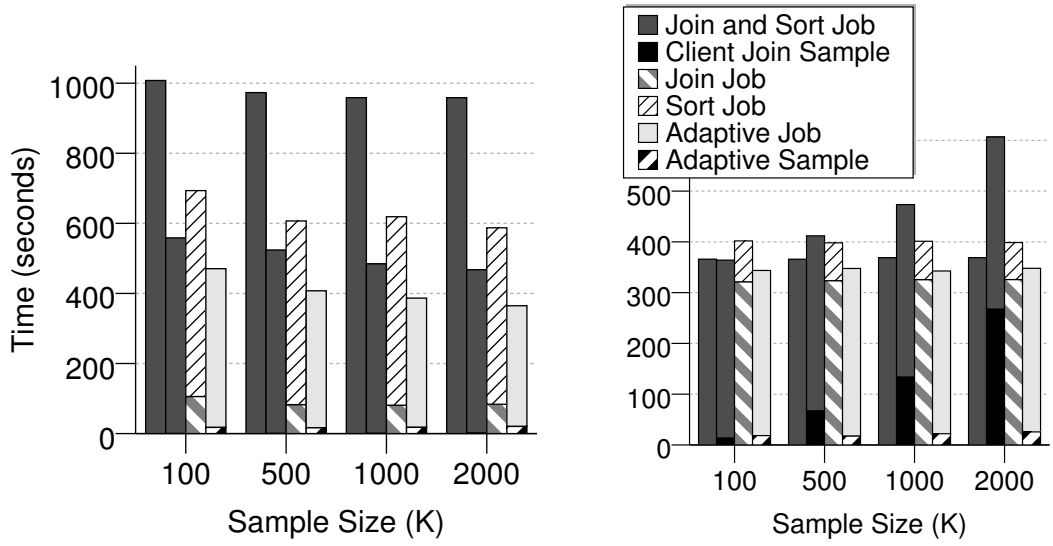
3. "Join Job" followed by "Client Input Sample" and "Sort Job": We use two MapReduce jobs where the first one computes the join while the second one sorts the data. After the first job finishes and before starting the second job, we sequentially sample some of the output of the first job and compute the partitioning map. We store the partitioning in HDFS and we use it in the second job.

4. "Adaptive Job": using AMs (four waves), AS, and AP.

Another alternative would be to use a sampling job followed by a join-and-sort job. This would be similar to the techniques available in Pig [57]. We believe that alternative (3) covers this case, as it also includes the overhead of using an additional job and would also require extra processing for computing the sample.

Notice that sampling based on the input data is not feasible if the sort key comes from the broadcast relation or is computed from both join inputs. We purposely chose to sort on the fact key to compare against client-input-sampling jobs. Also, alternative (2) assumes that client machine has enough resources to run the map function, i.e., that it has enough memory to load the reference data. In our case, we ran the client on the spare node in the cluster that Hadoop was using.

Due to the high volume of data being shuffled over the network and the large number of simultaneous readers of map outputs, we noticed that for these experiments some reducers reported network errors, resulting in very unstable performance. To avoid this problem we decreased the number of reduce slots by half.



(a) 100M records with a 1:30 fan-out and a cheap join predicate

(b) 10M records, 1:30 fan-out and an expensive join predicate

Figure 4.13: Performance comparison between regular sampling and partitioning versus adaptive sampling and partitioning on a JOIN-ORDER-BY query on the TERASORT dataset.

Figure 4.13(a) shows the running time of the entire query, including the time spent in the client, for the different methods with the number of collected samples varying from 100K to 2,000K samples. We used 100 million TERASORT records as input and

an average synthetic join fan-out of 1:30, i.e., for each input record, the join query produced an average of 30 results. Each of the four approaches is represented in the figure by a vertical bar (broken down by parts, if applicable). In this experiment, the time to sample the input data in the client is insignificant compared to the overall job time and is not visible in the figure. Notice that for the adaptive method the sampling time is visible in the figure but, the processing done while sampling is not lost. We can clearly see the benefit of sampling join results (alternative 2) versus sampling input data (alternative 1), as the input data did not capture the skew of the join result, leading to poor performance due to imbalance among the reducers. Even if sampling join results in the client did not take a significant amount of time, the Adaptive job (alternative 4) achieved a better performance than the client-join-sample job (alternative 2) due to the use of AMs. Because of the join fan-out, the map output increased $\times 30$ and AMs used all four waves of mappers (versus one used by the regular job) to overlap map time with shuffle time. The method using a join job and a sort job (alternative 3) did not perform very well because of the time spent on scheduling and starting an additional job. System logs for AS showed that for the 100K-sample case, around 60 mappers produced samples (out of 160 running in parallel), and it took around one second to elect the leader, collect and aggregate the samples, and propagate the partitioning map. For the 2,000K-sample case, the entire process took around three seconds.

Figure 4.13(b) demonstrates the danger of doing open-ended sampling in the client. In this experiment we used 10 million input records, and the same 1:30 fan-out. To emulate an expensive join, we introduced a look-up cost ($1ms$ per input record) and a match cost ($0.1ms$ per output pair). In this case the job sampling input data (alternative 1) was no longer significantly affected by the data skew, as its running time was dominated by the time spent in the map phase. The time spent in the client for computing join samples (alternative 2) increased with the sample size and becomes

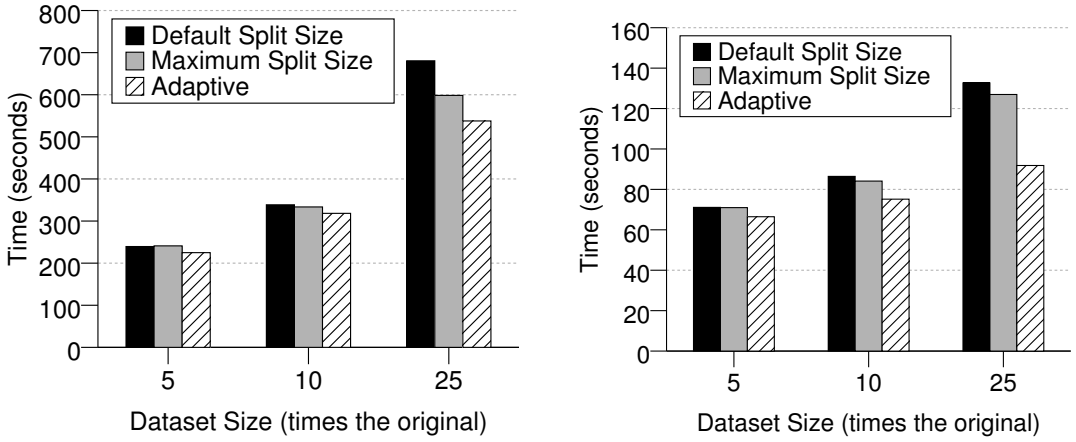
a significant part of the entire job. Again, the method using two jobs (alternative 3) did not perform very well due to the cost of running two MapReduce jobs. The adaptive method was the fastest of the four methods here as well.

We conclude that none of the client-sample methods, nor the multi-job method are a good overall solution, and that choosing a wrong method can affect performance negatively. On the other hand, using the adaptive techniques is a robust solution that offers good and stable performance. Overall, the adaptive job was seen to always be the fastest approach because it does not waste time in the client, it achieves a better balance in the reducers because it samples join results (not input data), and it allows the map time and the shuffle time to overlap.

4.4.4 Performance of End-to-End Set-Similarity Joins using Adaptive Techniques

In this section we evaluate the end-to-end performance of set-similarity joins using the adaptive techniques. We ran experiments using the same hardware and software setup as in Chapter 3. We performed a self-join on the DBLP dataset and a join between the DBLP and the CITESEERX datasets, using the same query parameters as in the previous chapter. We used the following setups: (1) regular Hadoop with a default split size, (2) regular Hadoop with maximum split size, and (3) Hadoop with AM and AC (Adaptive Hadoop). The regular Hadoop setting using the maximum split size (2) is equivalent to the setup in the previous chapter.

Figure 4.14(a) shows the running time for performing an end-to-end set-similarity self-join on the DBLP dataset increased between 5 and 25 times. We can see that Adaptive Hadoop brings modest performance improvements over regular Hadoop. To understand the overall benefit of the Adaptive Hadoop we looked at the performance



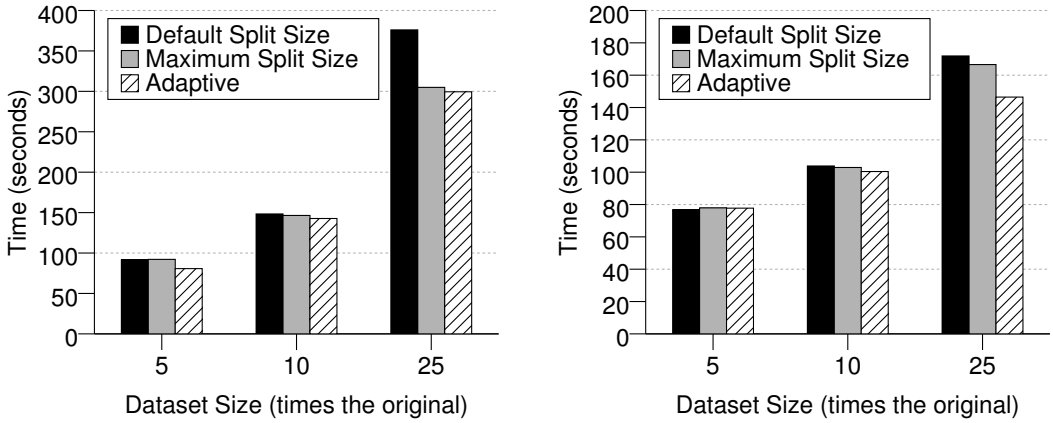
(a) End-to-End

(b) Stage 1: Basic Token Ordering (BTO)

Figure 4.14: Running time of end-to-end and Stage 1 for self-joining the DBLP $\times n$ dataset (where $n \in [5, 25]$) on a 10-node cluster.

characteristics for each stage. Figure 4.14(b) shows the running time for Stage 1 using the Basic Token Ordering (BTO) alternative. We can see that Adaptive Hadoop brings almost a 30% improvement over regular Hadoop. This is mainly due to the use of AC which groups the tokens using a hash-table. Figure 4.15(a) shows the running time for Stage 2 using the Indexed Kernel (PK) alternative. We can see that the benefits of Adaptive Hadoop are negligible in this case. Figure 4.15(b) shows the running time for Stage 3 using Basic Record Join. We can see that the adaptive techniques bring a modest benefit.

Figure 4.16 shows the running time for performing a end-to-end set-similarity join between DBLP and CITESEERX datasets increased between 5 and 25 times. We can see the modest benefit that Adaptive Hadoop brings over regular Hadoop. To further understand the benefit of Adaptive Hadoop we looked at the performance characteristics of each individual stage. Stage 1 is identical to the Stage 1 for self-join. Figure 4.17(a) shows the running time for Stage 2 using Indexed Kernel (PK). We can see that Adaptive Hadoop brings a modest benefit in this stage. Figure 4.17(b) shows the running time for Stage 3 using Basic Record Join (BRJ). We can see that



(a) Stage 2: Indexed Kernel (PK)

(b) Stage 3: Basic Record Join (BRJ)

Figure 4.15: Running time of stages 2, and 3 for self-joining the DBLP $\times n$ dataset (where $n \in [5, 25]$) on a 10-node cluster.

the adaptive techniques bring an almost 20% benefit over regular Hadoop using the maximum split size. This is due to AM which better balanced the input splits across map tasks.

4.4.5 Performance Summary of Adaptive Techniques

Based on our performance experiments, we can make the following observations:

- The adaptive techniques offer a modest performance benefit for end-to-end set-similarity joins.
- For other important classes of queries, the adaptive techniques were seen to offer significant performance improvements - sometimes providing $\times 3$ improvements - over the best regular MapReduce job.
- Even if mis-configured, the adaptive techniques were never seen to hurt the performance of the MapReduce jobs (with the one exception of AC's LRU cache policy for small cache sizes).

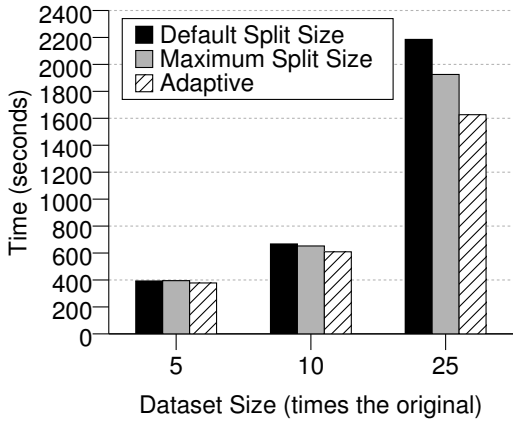
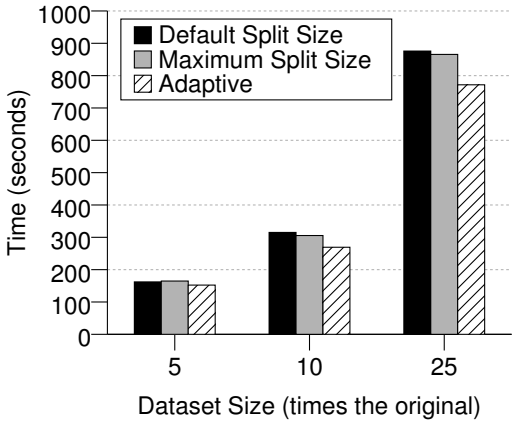


Figure 4.16: Running time for joining the DBLP $\times n$ and CITESEERX $\times n$ dataset (where $n \in [5, 25]$) on a 10-node cluster.

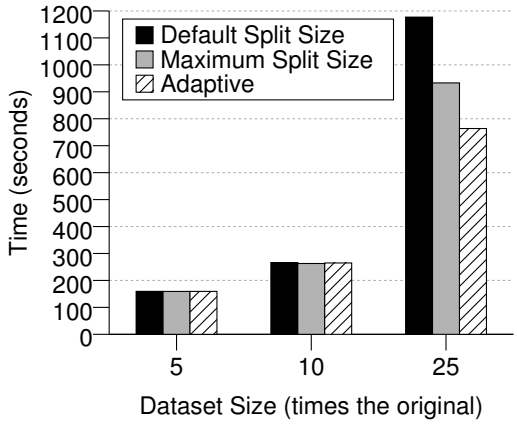
4.5 Conclusions

In this chapter we have presented a number of techniques to improve set-similarity join performance by making the MapReduce execution model more adaptive. These techniques are general in nature, and thus also help the performance of other common queries. These adaptive techniques utilize "Situation-Aware Mappers" that are able to cooperatively make global optimization decisions. From our experimental evaluation, we observed that the adaptive techniques bring modest performance improvements for end-to-end set-similarity joins For other important classes of queries, the adaptive techniques can bring significant performance improvements. The adaptive techniques never hurt the performance of set-similarity joins (or other queries) the MapReduce jobs with the exception of the AC's LRU cache policy for small cache sizes.

We believe that SAMs could become an important extension point in the MapReduce execution framework. Advanced users can implement what are essentially system-level enhancements using this mechanism, just as we implemented the adaptive techniques described in this chapter.



(a) Stage 2: Indexed Kernel (PK)



(b) Stage 3: Basic Record Join (BRJ)

Figure 4.17: Running time of stages 2 and 3 for self-joining the DBLP $\times n$ and CITESEERX $\times n$ dataset (where $n \in [5, 25]$) on a 10-node cluster.

Chapter 5

Efficient Set-Similarity Joins in ASTERIX

5.1 Introduction

With the increasing need for set-similarity join operations in various applications, it would be beneficial to have a set-similarity join operator at a system structure level, i.e., so a user can write a set-similarity join by writing a few lines in a declarative language, instead of writing hundreds of lines of code. In this way, expressing a set-similarity join operation would become easy for application developers, analysts, and statisticians. Moreover, set-similarity join operations could be applied on various types of data, including strings, sets, images, etc. without making changes to the code. A general set-similarity operator available at a query language feature level will allow a smooth integration with other data types by providing hooks for data-type-specific tokenizers and similarity functions. Finally, a set-similarity operator at this level would allow for more complex fuzzy-join queries that could, for example,

include top-k or multi-way set-similarity join queries.

We chose to study how to integrate set-similarity joins as a first-class operator in the ASTERIX parallel data-intensive computing system. We choose ASTERIX as our underlying system for set-similarity joins for the following reasons: ASTERIX is based on Hyracks data-intensive runtime platform. Hyracks provides native support for joins and has proven to provide superior performance over Hadoop [14]. While a number of other high-level languages are built on top of Hadoop, such as Hive [59], Pig [51] and Jaql [12], and could thus run on Hyracks using Hyracks' compatibility layer, ASTERIX's AQL language is the only one that fully leverages Hyracks' performance characteristics. Moreover, compared with similar systems such as Dryad [39] and Scope [17], Hyracks and ASTERIX are available in open-source. Nevertheless, the techniques presented in this chapter could be adapted to other systems using a rule-based compiler and a more general runtime than MapReduce.

In this chapter, we show which are the touch points along the various layers of the system (including parser, optimizer and runtime) and what are the necessary changes to do the integration of fuzzy joins in ASTERIX. The contributions of this chapter are as follows:

- We conduct a design study where we implement our three-stage set-similarity join algorithm on the ASTERIX runtime platform, namely Hyracks.
- We analyze the relative performance of our Hadoop and Hyracks implementations of the fuzzy-join algorithm.
- We design the AQL language syntax for expressing fuzzy joins. We propose a simple syntax with predefined values for a similarity function and a threshold, and a more detailed syntax that allows for specifying the similarity function and threshold individually for each fuzzy predicate.

- We present the techniques necessary to translate a fuzzy-join query into a Hyracks plan that incorporates the three-stage set-similarity join algorithm.
- We design a compiler-level language, aimed at expressing complex rewriting rules, that reuses the ASTERIX parser and translator. This language allows us to easily specify the rewriting rule for set-similarity joins and it is general enough to also be useful in other cases.
- We conduct a performance study in which we compare the performance of two set-similarity join plans in Hyracks: a plan generated by the ASTERIX compiler, and a hand-written plan.

This chapter is organized as follows. In Section 5.2 we study the feasibility of implementing set-similarity joins in the ASTERIX's runtime platform. In Section 5.3 we describe the integration of set-similarity join queries into the ASTERIX high-level query language and its implementation. Then, in Section 5.4 we conduct our verification study, where we analyze the relative performance of the hand-coded fuzzy-join plan with the plan generated by the ASTERIX compiler. We conclude this chapter in Section 5.5.

5.2 Design Study Comparison

Before we set-off to implement fuzzy-joins in ASTERIX, we first studied the feasibility of implementing fuzzy-joins in ASTERIX's runtime, Hyracks. In this section we provide more details about Hyracks' runtime, including its MapReduce compatibility mode, propose an implementation of the three-stage fuzzy-join algorithm in Hyracks, and conduct a performance comparison between the Hadoop and Hyracks implementations.

5.2.1 Hyracks Computational Model

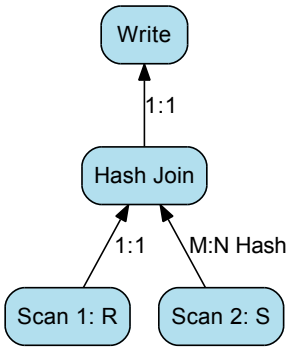


Figure 5.1: A Hyracks plan for a join between two datasets R and S

Hyracks is a partitioned-parallel dataflow execution platform that runs on shared-nothing clusters of computers. Large collections of data items are stored as local partitions that are distributed across the nodes of the cluster. A Hyracks job (the unit of work in Hyracks), submitted by a client, processes one or more collections of data to produce one or more output collections (also in the form of partitions). Hyracks provides a programming model and an accompanying infrastructure to efficiently divide computations on large data collections (spanning multiple machines) into computations that work on each partition of the data separately. A Hyracks job is a dataflow DAG composed of operators (nodes) and connectors (edges). Operators represent the job's partitioned-parallel computation steps, and connectors represent the (re-) distribution of data from step to step. Figure 5.1 shows an example of a simple Hyracks job for a join between two datasets R and S. The job uses a hash join operator and hash partitions the S dataset (using an "`M-N Hash`" connector). In this example, we assume that the R dataset does not need to be repartitioned as it is already partitioned on the join column.

In Hyracks, data flows between operators over connectors in the form of records (or tuples) that can have an arbitrary number of fields. Hyracks provides support for expressing data-type-specific operations such as comparisons and hash functions. The

Hyracks use of a record as the carrier of data is a generalization of the (key, value) concept found in MapReduce and Hadoop. The advantage of the generalization is that operators do not have to artificially package (and repackage) multiple data objects into a single "key" or "value" object.

Hyracks has been designed with the goal of being a runtime platform where users can hand-create jobs, as in Hadoop and MapReduce, yet at the same time being an efficient target for the compilers of higher-level programming languages such as AQL (ASTERIX), Pig, Hive, or Jaql.

One of the initial design goals of Hyracks has been for it to be an extensible runtime platform. To this end, Hyracks includes a rich API for operator implementers to use when building new operators. The implementations of the operators made available "out of the box" via the Hyracks operator library are also based on the use of this same API.

5.2.2 MapReduce Compatibility Mode

To facilitate easy migration for Hadoop users who wish to use Hyracks, a Hadoop compatibility layer on top of Hyracks is available. The aim of the Hyracks Hadoop compatibility layer is to accept Hadoop jobs unmodified and run them on a Hyracks cluster. Hyracks provides two extra operators that can wrap the `map` and `reduce` functionality.[1] The data tuples provided as input to the *hadoop_mapper* operator are treated as `(key, value)` pairs and, one at a time, passed to the user-provided `map` function. The *hadoop_reducer* operator also treats the input as `(key, value)` pairs, and groups the pairs by `key`. The sets of `(key, value)` pairs that share the same `key` are passed, one set at a time, to the user-provided `reduce` function. The

[1] The `combine` functionality of MapReduce is fully supported as well, but those details are omitted here for ease of exposition.

outputs of the `map` and `reduce` functions are directly output by the operators. To emulate a MapReduce framework, Hyracks employs a sort operator and a hash-based distributing connector.

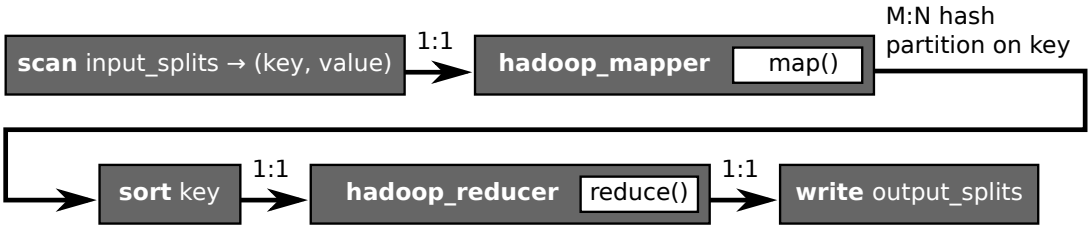


Figure 5.2: A Hyracks plan for a Hadoop job

Figure 5.2 shows the whole pre-configured Hyracks plan for running a MapReduce job. After data is read by the scan operator, it is fed into the *hadoop_mapper* operator using a 1:1 edge. Next, using an M:N hash-based distribution edge, data is partitioned based on `key`. (The "M" value represents the number of mappers, while the "N" value represents the number of reducers.) After distribution, data is sorted using the sort operator and passed to the *hadoop_reducer* operator using a 1:1 edge. Finally, the *hadoop_reducer* operator is connected using a 1:1 edge to a file-writer operator.

Our first approach to implementing set-similarity joins in Hyracks is to use the Hyracks' MapReduce compatibility layer and the MapReduce code developed in Chapter 3. The jobs comprising each of our three stages are given as input to the Hyracks compatibility layer which, in turn, runs them on Hyracks using the plan from Figure 5.2. For example, in Stage 1, the Basic Token Ordering (BTO) approach consists of two MapReduce jobs. The processing works as follows: The first MapReduce job is given to the Hyracks compatibility layer, which in turn schedules a Hyracks job using the Map and Reduce code from the original MapReduce job. After the first Hyracks job is finished, the output is written to disk, and the second MapReduce job of Stage 1 is ran in a similar fashion. The second Hyracks job reads from disk the output of the first Hyracks job. The wrapping of Hadoop MapReduce jobs into Hyracks

jobs is done automatically by the compatibility layer with no user intervention.

Our second approach to implementing set-similarity joins in Hyracks is to rewrite our code from scratch using Hyracks' native operators, instead of being limited to only Map and Reduce wrappers. We discuss this approach next.

5.2.3 Set-Similarity Joins in Hyracks

In this section we propose a native implementation of the three-stage set-similarity join algorithm in Hyracks. We use the rich library of operators and connectors available in Hyracks, as we are no longer constrained to using only the "`map`" and "`reduce`" functions from the MapReduce framework. Moreover, for some operations, we build our own custom operators. Finally, we implement the three-stage algorithm as a single Hyracks job instead of three MapReduce jobs.

Figure 5.3 shows the Hyracks job specification to natively implement the three stages of our parallel set-similarity join algorithm between two datasets, "R" and "S". The plan is represented bottom-up and the three stages are marked in the figure by gray boxes.

Stage 1: Token Ordering. The plan starts with a scan of one of the datasets, "R". The first operator in this stage is a custom operator, "`Extract Tokens`", that tokenizes the join column and outputs one tuple per token. Next, the tokens are aggregated and the frequency of each token is computed. The aggregation is first conducted locally (via the first "`Hash Group`" operator) and then globally (via the second "`Hash Group`" operator) using an "`M:N Hash`" connector for hash partitioning the tokens. After the counts are computed, the tuples are locally-sorted by count and merged in a sorted order at one node using an "`M:N Hash Merge`" connector (where

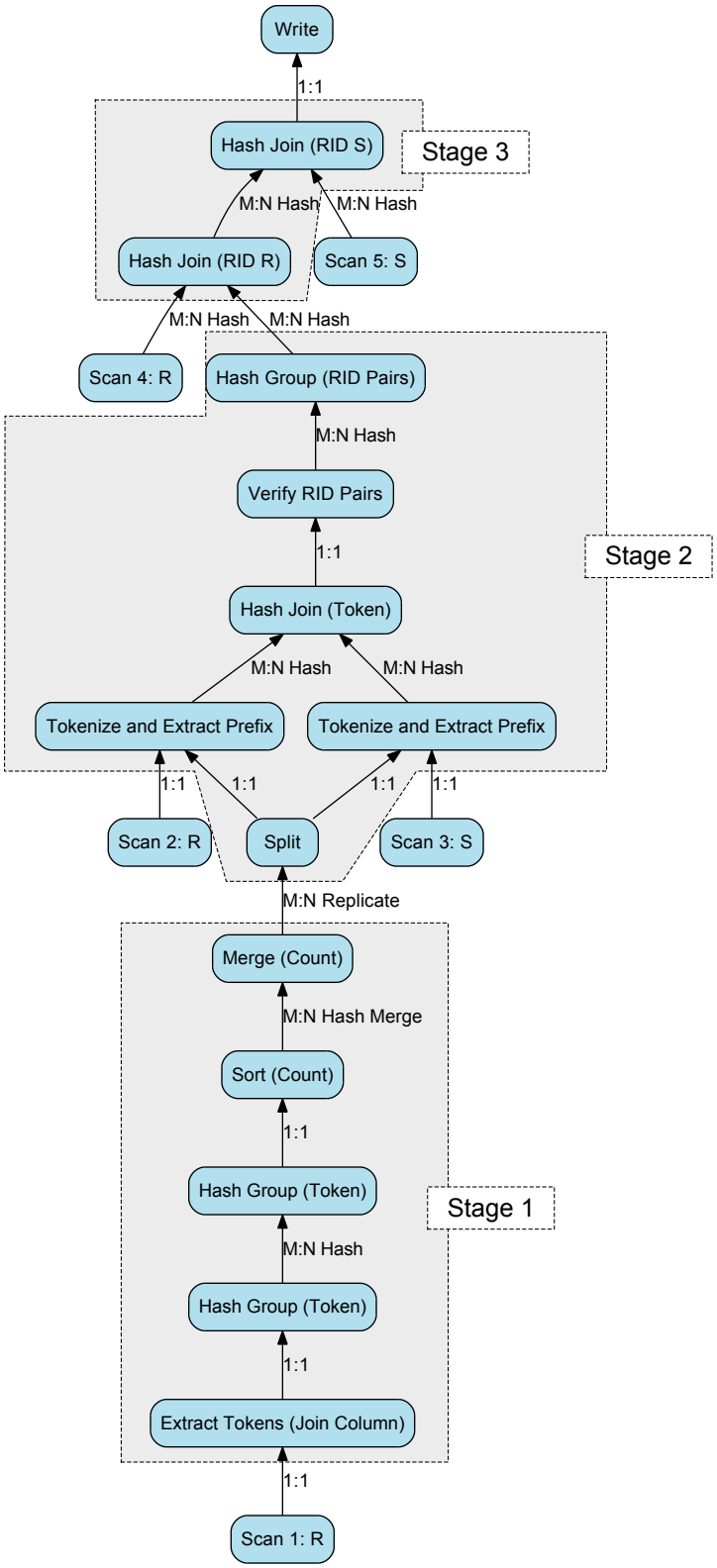


Figure 5.3: Hyracks-native plan for Set-Similarity Join.

N=1) and a "Merge" operator.

Stage 2: RID-Pair Generation. The second stage, RID-Pair Generation, gets as input the two datasets and the ordered tokens produced in the first stage. The tokens are replicated from the node that produced them in Stage 1 to all the nodes that participate in Stage 2. This is achieved using an "M:N Replicating Connector" and a "Split" operator. The token streams and the original inputs are provided as input to a custom operator, "Tokenize and Extract Prefix". This operator tokenizes the join column of each input record, orders the record's tokens by the global order, and extracts the prefix tokens. The processing is very similar to the processing in the map functions of the second stage of the MapReduce implementation (see Chapter 3). The operator outputs one tuple per prefix token. Each tuple also contains the RID and the value of the join column. This happens in parallel for the two input datasets. The two streams of prefix tokens and record projections are hash partitioned and joined on tokens using an "M:N Hash" connector and a "Hash Join" operator. Each of the pairs produced by the join operator passes the prefix filter and is verified to remove false positives using another custom operator, "Verify RID Pairs". Finally, this stage ends with a "Hash Group" operator where duplicate RID pairs produced due to redistribution by prefix tokens are removed.

Stage 3: Record Join. The third stage, Record Join, consists of two join operations. The first operation joins the records from the first dataset with the RID pairs produced in the second stage. The output is a set of half-filled join pairs. The second half is filled by the second join operation that joins the half-filled pairs with the second dataset. Finally, the join result is written to the output.

Figure 5.4 shows the Hyracks job for a set-similarity self-join. The differences between the R-S join case and the self-join case are mainly in Stage 2. In the self-join case we use a "Hash Group" operator, instead of a "Hash Join" operator, to find pairs

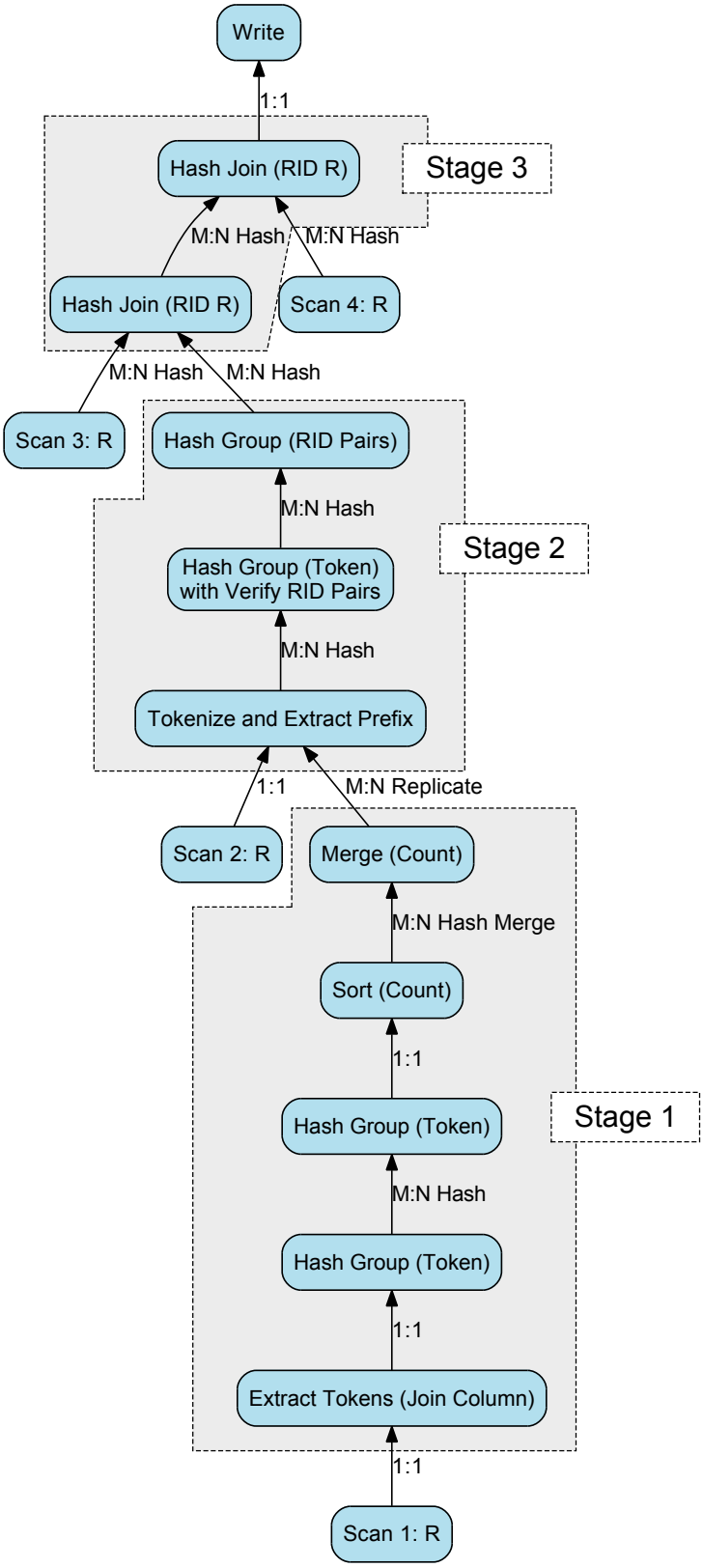


Figure 5.4: Hyracks-native plan for Set-Similarity Self-Join.

of records that share a token in common in the prefix. Moreover, the function that verifies the RID pairs is applied as an aggregator for the group-by operator. For stages 1 and 3, the R-S join plan and the self-join plan are very similar.

5.2.4 Performance Comparison

In this section we compare the Hadoop-based and the Hyracks-based implementations of our set-similarity join algorithms. For the Hadoop-based implementation, we take the best approach as described in Chapter 3. That is, we use Basic Token Ordering (BTO) for Stage 1, Indexed Kernel (PK) for Stage 2, and Basic Record Join (BRJ) for Stage 3. In order to compare the performance on a stage-by-stage basis, we split the plan for the Hyracks-native implementation into three individual parts, one for each stage.

We ran experiments on the same 10-node IBM cluster described in Chapter 3. The Hadoop setup was the same as the one used in Chapter 3. For Hyracks, we ran one Node Controller process on each node and one Cluster Controller process on the master node. On each Node Controller, we ran four instances of each Hyracks operator. This is roughly equivalent to having four Map or Reduce slots in a Hadoop cluster. We computed a set-similarity self-join of the DBLP dataset on publication titles and authors. That is, we were looking for publications with similar titles written by a similar set of authors. We increased the DBLP data sizes to $\times 5$, $\times 10$, and $\times 25$ times the original dataset, as described in Chapter 3.

Table 5.1 shows the running times for the three stages of the set-similarity join algorithm on the three different sizes of DBLP data. The row labeled "Hadoop" represents the time taken to run each stage as a Hadoop job; "Compatibility" shows the time taken to run the same MapReduce jobs using the Hadoop compatibility layer on

Hyracks; finally, "Native Hyracks" represents the time taken to run the Hyracks-native implementation of the algorithms.

Data Size	Platform	Stage 1	Stage 2	Stage 3	Total time (seconds)
×5	Hadoop	75.52	103.23	76.59	255.34
	Compatibility	24.50	38.92	23.21	86.63
	Native Hyracks	11.41	32.69	9.33	53.43
×10	Hadoop	90.52	191.16	107.14	388.82
	Compatibility	42.15	128.86	38.47	209.49
	Native Hyracks	16.14	81.70	14.87	112.71
×25	Hadoop	139.58	606.39	266.26	1012.23
	Compatibility	105.98	538.74	60.62	705.34
	Native Hyracks	34.50	329.30	27.88	391.68

Table 5.1: Running times for parallel set-similarity joins for different dataset sizes on a 10 node cluster.

In the column labeled "Stage 1" we see that the MapReduce jobs running on Hadoop scaled smoothly with increasing data sizes. The Hadoop compatibility layer on Hyracks and the Native Hyracks formulation also show this trend. However, the jobs running on Hyracks were faster because they benefited from two improvements: low job startup overhead and more efficient data movement by the infrastructure. Hyracks uses push-based job activation, which introduces very little wait time between the submission of a job by a client and the time when it begins running on the node controllers. Additionally, the Native Hyracks formulation uses hash-based grouping, so it does not need to sort the input data like in Hadoop. This reduces the running time even more.

In the column labeled "Stage 2" we observe a near constant difference between the running times of Hadoop and Hyracks compatibility-mode. This is because most of the time in this stage is spent in the Reducer code that uses an algorithm with a quadratic time complexity to find matching records. The details of this algorithm were presented in Chapter 3. The key observation is that most of the work in this stage is done in the Reduce phase in user-defined code, so the only improvement we

can expect is due to lower overhead of the infrastructure. For the Native Hyracks formulation, we used sort-based grouping since it performed slightly better than its hash-based counterpart. Hash-based aggregation usually works well for traditional aggregates when grouping is performed on relatively low-cardinality attributes, as compression is achieved by the aggregation. In the case of Stage 2, however, the grouping operation is used to separate the input into groups without applying an incremental aggregation function. Moreover, the sort operator in Hyracks implements a "Poor man's normalized key" optimization [32] that can be exploited when building Hyracks Native jobs; this optimization helps in terms of providing cheaper comparisons and better cache locality. After sorting, the "aggregation" function computes the Cartesian product of the items within each group. This shows up in Table 5.1 as a super-linear growth in the completion time for Stage 2, Native Hyracks, due to larger group sizes.

The column labeled "Stage 3" in Table 5.1 shows that at the $\times 5$ data size, the ratio of the Hadoop completion time to the Hyracks compatibility layer completion time was 3.3, and the ratio drops to about 2.8 at the $\times 10$ data size. As the amount of data grows to $\times 25$, we see the compatibility layer completing about 4 times faster than Hadoop. This is because Stage 3 performs two "joins" of the record-ids with the original data to produce similar-record pairs from the similar-record-id pairs that were produced in the previous stage. Every join step reshuffles all of the original input data from the Map to the Reduce phase, but the output of the join is fairly small. Most of the work is thus performed in moving the data (as the Map and Reduce code is fairly simple). At smaller data sizes, Hadoop times were dominated by the startup overhead, while for larger data sizes, the dominating factor is data redistribution. The Native Hyracks case benefits from using hash-join operators that perform the joins by building a hash-table on one of their inputs and then probing the table using the other input. This strategy is cheaper than first sorting both inputs on the join

key to bring the matching values from both sides together (which is what happens in Hadoop).

In summary, we see that the Native Hyracks formulation of Set-Similarity Joins ran about 2.6 to 4.7 times faster than their MapReduce formulation running on Hadoop and about 1.6 to 1.8 times faster than running the same MapReduce jobs on Hyracks using its compatibility layer. These positive results motivated us to implement fuzzy-join as a first-class operator in ASTERIX.

5.3 Implementation in AQL

In this section we describe our set-similarity join implementation in ASTERIX's high level query language, AQL. We start with an introduction to the ASTERIX user model. Next, we describe our proposed syntax for expressing fuzzy-joins in AQL. We end this section with a description of the necessary steps for transforming a fuzzy-join AQL query into an efficient Hyracks-based execution plan.

5.3.1 ASTERIX User Model

The user model of ASTERIX consists of its data model (ADM) and query language (AQL) targeting semistructured data. We will illustrate the features of ADM and AQL via a few examples using the two publication datasets introduced in Chapter 1, DBLP and CITESEERX.

The ASTERIX data model (ADM) is inspired by the features of popular formats for semistructured data, such as JSON [44] and Avro [2], as well as by the structured data types of object-database systems [54] from which ADM gets bulk types such as

sets. We call an instance of the ASTERIX data model a *value*. The type of a value can either be a *primitive* type (int32, int64, string, time, etc., or null) or a *derived* type, that may include: Enum, Record, Ordered List, Unordered List, and Union.

Figure 5.5 shows one possible way to model the two publication datasets from our example. DBLP publications are modeled as a `DBLPType` record with fields such as the `id`, `dblpid`, `title`, `authors`, and a `miscellaneous` field. The DBLP record type is indicated as open to allow additional structured content to be associated with a publication instance. The CITESEERX publications are modeled in a similar fashion using the `CSXType` record type.

```
declare type DBLPType as open {
  id: int32,
  dblpid: string,
  title: string,
  authors: [string],
  miscellaneous: string
}
```

(a)

```
declare type CSXType as open {
  id: int32,
  csxid: string,
  title: string,
  authors: [string],
  miscellaneous: string
}
```

(b)

```
create dataverse Publications;
use dataverse Publications;

create dataset DBLP(DBLPType) partitioned by key id;
create dataset CSX(CSXType) partitioned by key id;
```

(c)

Figure 5.5: Metadata definition for the running example

A named collection of data in ASTERIX is called a *dataset*. A dataset may be partitioned over multiple nodes in a cluster. Partitions could be replicated on other nodes to allow parallel access as well as to provide fault-tolerance. A collection of datasets related to an application are grouped into a namespace called a *dataverse*. Figure 5.5(c) shows the DDL (data definition language) commands to create a dataverse "Publications" with datasets "DBLP" and "CSX". The DDL includes the specification of a primary key for each dataset (currently required for all datasets) as well as

a partitioning key for the system to use in logically distributing the data across the nodes of a cluster. By default, the partitioning key is also the primary key.

In Figure 5.6, we exemplify our data model by showing a few records that could appear in the DBLP dataset. Datasets are always unordered collections, and we denote this in the figure by using angular brackets.

```
< {
    "id": 1023,
    "dblpid": "conf/icip/SchonfeldL98",
    "title": "VORTEX  Video Retrieval and Tracking from Compressed
Multimedia Databases.",
    "authors": [ "Dan Schonfeld", "Dan Lelescu" ],
    "miscellaneous": "1998 ICIP (3)",
  },
  {
    "id": 2403,
    "dblpid": "books/acm/kim95/AnnevelinkACFHK95",
    "title": "Object SQL - A Language for the Design and Implementation of
Object Databases.",
    "authors": [ "Jurgen Annevelink", "Rafiul Ahad", "Amelia Carlson",
"Daniel H. Fishman", "Michael L. Heytens", "William Kent" ],
    "miscellaneous": "1995 Modern Database Systems",
  } >
```

Figure 5.6: Data from the DBLP dataset

The ASTERIX Query Language (AQL) borrows from XQuery 1.0 [65] and Jaql 0.1 [41] their programmer-friendly declarative syntax that describes bulk operations such as iteration, filtering, and sorting. As a result, AQL is comparable to those languages in terms of expressive power. We use examples to illustrate key features of the AQL language. The examples are asked against records stored in the publication datasets defined earlier. Datasets can be accessed using the built-in "`dataset`" function. For instance, the function call `dataset('DBLP')` exposes a collection of publications of "DBLP" type.

Q1. **(Simple AQL Scan)** List all DBLP titles:

```
for $dblp in dataset('DBLP')
return $dblp.title
```

Note that AQL uses the classic dot "." syntax for field access.

Q2. **(Group-by and Order-by in AQL)** List DBLP authors in decreasing order of their publication count:

```
for $dblp in dataset('DBLP')
let $id := $dblp.id
for $author in $dblp.authors
group by $dblp_author := $author with $id
order by count($id) desc
return $dblp_author
```

This query starts by looping over the DBLP publications. For each publication, it loops over the author names. The "`group by`" re-binds the "$id" variable via its "`with`" clause to the list of DBLP paper ids grouped under the same author name. The list is passed, to the "`count`" aggregate, for sorting the output.

Q3. **(Join in AQL)** Find all pairs of publications with identical titles from the two datasets:

```
for $dblp in dataset('DBLP')
for $csx in dataset('CSX')
where $dblp.title = $csx.title
return {'dblp': $dblp, 'csx': $csx}
```

This query returns a collection of a new type of record that contains two fields, one for storing the DBLP publications and one for storing the CITESEERX publications.

5.3.2 Set-Similarity-Join Syntax in AQL

We introduce the AQL set-similarity-join syntax through a couple of examples.

FJ1. **(Fuzzy-Join in AQL)** Find all pairs of publications with similar titles from the two publication datasets. The similarity between titles is measured using the Jaccard similarity function. Two titles are deemed similar if their Jaccard similarity is greater or equal than 0.9:

```
set simfunction 'jaccard';
set simthreshold '0.9';

for $dblp in dataset('DBLP')
for $csx in dataset('CSX')
where $dblp.title ~= $csx.title
return { 'dblp': $dblp, 'csx': $csx }
```

Note that before the actual query we specify the similarity function and threshold using two "`set`" statements. The similarity predicate is expressed using the "~=" operator.

FJ2. **(Top-k Fuzzy-Join in AQL)** Find the top-10 pairs of publications with similar sets of authors from the two datasets that have at least a 0.9 Jaccard similarity:

```
for $dblp in dataset('DBLP')
for $csx in dataset('CSX')
let [$match, $sim] :=
    $dblp.authors ~= $csx.authors
    with simfunction  'jaccard', simthreshold '0.9'
where $match
order by $sim desc
```

```
limit 10
return { 'dblp': $dblp, 'csx': $csx, 'similarity': $sim }
```

Note the slightly different syntax around the "~=" operator. In this case, we specify the similarity function and threshold inline with the operator. In general, this allows for multiple similarity operators with different similarity functions or thresholds in the same query. Moreover, the extended predicate syntax includes the return of two values: a boolean value "`match`" that is true only if the pair of titles passes the similarity threshold, and a numeric value "`sim`" that holds the value of the pair's similarity. The "`sim`" value is used later to order the pairs by similarity and then limit the results to only the highest 10.

5.3.3 Set-Similarity Join Processing in AQL

The AQL compiler is rule-based and has various rules for detecting joins, pushing projections, etc. To be able to execute fuzzy-join queries efficiently in AQL, we have added a fuzzy-join rule. The goal of the rule is to rewrite a join query with a fuzzy predicate ("~=") into a logical plan that incorporates the three stages of our fuzzy-join algorithm.

Before going into the details of how the fuzzy-join rule rewrites the logical join into our three-stage algorithm, we first discuss how our algorithm can actually be expressed in AQL. Using the AQL syntax, Figure 5.7 shows an AQL query for our three-stage algorithm for a fuzzy join of the DBLP and CITESEERX datasets on the "title" column using Jaccard similarity and a threshold of 0.9. Figure 5.8 shows the Hyracks plan for this query. The three stages are marked in the figure by gray boxes. Next, we discuss the details of the AQL subquery for each stage and its corresponding Hyracks subplan.

```
1  // -- - Stage 3 - --
2  for $paperCSX in dataset('CSX')
3  for $paperDBLPridpair in
4      for $paperDBLP in dataset('DBLP')
5      for $ridpair in
6          // -- - Stage 2 - --
7          for $paperDBLP in dataset('DBLP')
8          let $idDBLP := $paperDBLP.id
9          let $tokensUnrankedDBLP := counthashed-word-tokens($paperDBLP.title)
10         let $lenDBLP := len($tokensUnrankedDBLP)
11         let $tokensDBLP :=
12             for $tokenUnranked in $tokensUnrankedDBLP
13             for $tokenRanked at $i in
14                 // -- - Stage 1 - --
15                 for $paper in dataset('DBLP')
16                 let $id := $paper.id
17                 for $token in counthashed-word-tokens($paper.title)
18                 /*+ hash */
19                 group by $tokenGrouped := $token with $id
20                 /*+ inmem */
21                 order by count($id), $tokenGrouped
22                 return $tokenGrouped
23             where $tokenUnranked = /*+ bcast */ $tokenRanked
24             order by $i
25             return $i
26         for $prefixTokenDBLP in subset-collection(
27                                 $tokensDBLP, 0, prefix-len-jaccard(len($tokensDBLP), 0.9))
28
29         for $paperCSX in dataset('CSX')
30         let $idCSX := $paperCSX.id
31         let $tokensUnrankedCSX := counthashed-word-tokens($paperCSX.title)
32         let $lenCSX := len($tokensUnrankedCSX)
33         let $tokensCSX :=
34             for $tokenUnranked in $tokensUnrankedCSX
35             for $tokenRanked at $i in
36                 // -- - Stage 1 - --
37                 // same subquery as "Stage 1" above
38             where $tokenUnranked = /*+ bcast */ $tokenRanked
39             order by $i
40             return $i
41         for $prefixTokenCSX in subset-collection(
42                                 $tokensCSX, 0, prefix-len-jaccard(len($tokensCSX), 0.9))
43
44         where $prefixTokenDBLP = $prefixTokenCSX
45         let $sim := similarity-jaccard-prefix(
46                         $lenDBLP, $tokensDBLP, $lenCSX, $tokensCSX, $prefixTokenDBLP, 0.9)
47         where $sim >= 0.9
48         /*+ hash */
49         group by $idDBLP := $idDBLP, $idCSX := $idCSX with $sim
50         return {'idDBLP': $idDBLP, 'idCSX': $idCSX, 'sim': $sim[0]}
51
52     where $ridpair.idDBLP = $paperDBLP.id
53     return {'idCSX': $ridpair.idCSX, 'paperDBLP': $paperDBLP, 'sim': $ridpair.sim}
54
55 where $paperDBLPridpair.idCSX = $paperCSX.id
56 return {'dblp': $paperDBLPridpair.paperDBLP, 'csx': $paperCSX}
```

Figure 5.7: Three-stage set-similarity join algorithm expressed in AQL for a join between the DBLP and CITESEERX datasets using the Jaccard similarity and a threshold of 0.9.

Stage 1: Token Ordering

The first stage, Token Ordering, is expressed in lines 14-22 in Figure 5.7. In this subquery, we iterate over the records in the DBLP dataset and, for each record, we extract the tokens from the title using the "`counthashed-word-tokens`" function. This function is one of the tokenizers available in AQL and outputs a list of integers that correspond to the hash-codes of the tokens in the input string, where multiple appearances of the same token are considered as new tokens. Next, we unnest the token list and we count the number of occurrences of each token using a group-by statement. For the "`group by`" statement we introduce a compiler hint in line 18 that recommends the use of hash-based aggregation instead of the default sort-based aggregation. Finally, we order the tokens by count. Again, we use a compiler hint in line 20 to mark the fact that the aggregated list of tokens should fit in memory. This fact will be used later in Stage 2. Ideally, this fact would be detected by Hyracks, at runtime, after the sort has been completed. The same subquery is repeated later, in lines 36-37, in the context of the second dataset. The compiler will detect the duplicate subquery, and it will execute the subquery only once and use a "`Split`" runtime operator to send the results to both outer plans.

Hyracks Plan: When this AQL subquery is translated to a Hyracks plan, the "`for`" loop in line 15 becomes the "`Scan 1: DBLP`" operator in Figure 5.8. The "`counthashed-word-tokens`" function call and the "`for`" loop in line 17 are captured in the "`Meta 1: Tokenize Join Column, Unnest`" operator. Meta operators are collections of light-weight, in-memory, streaming, co-located operators that communicate through main memory and do not need connectors between them. In this case the "`Meta 1`" operator contains two such light-weight operators, "`Tokenize Join Column`" and "`Unnest`". The "`group by`" statement in line 19 becomes two "`Hash Group`" operators: the first one (in the bottom-up order) is a local group-by, while the second

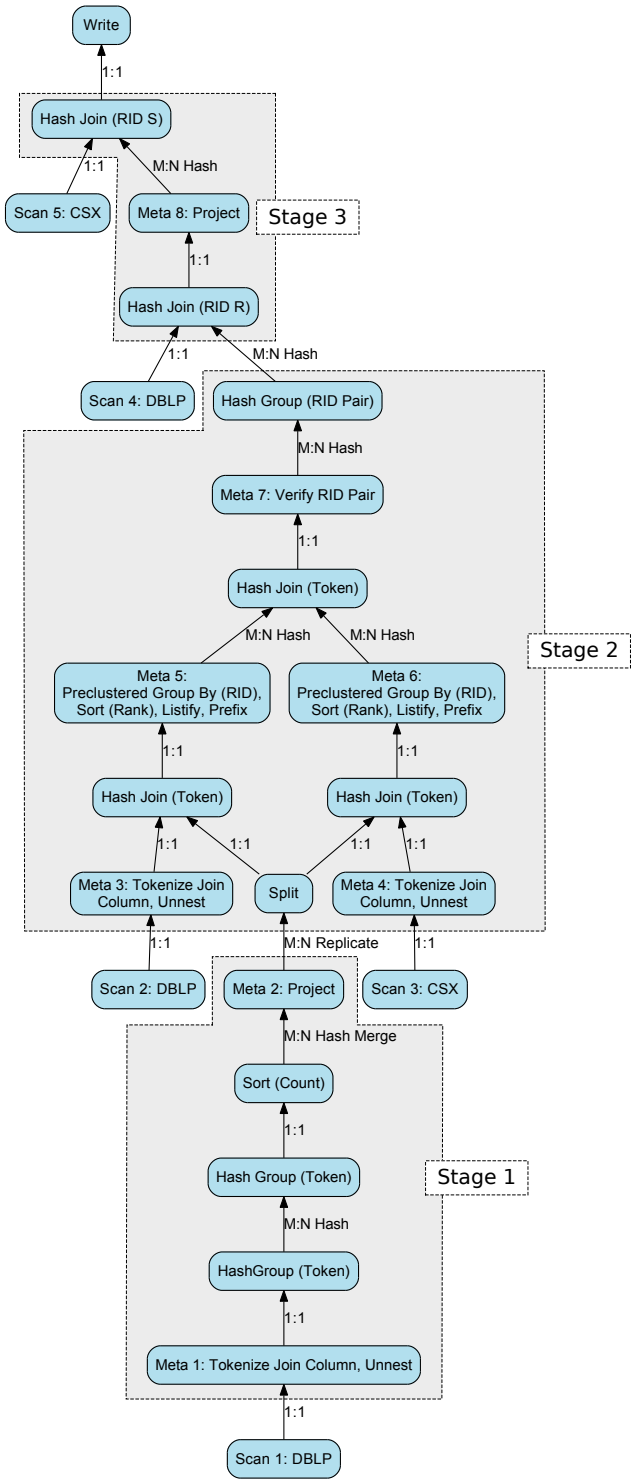


Figure 5.8: The Hyracks physical plan for the AQL Set-Similarity Join query in Figure 5.7.

one is a global group-by. The group-by is followed by a local sort and a merge at one node using an "`M:N Hash Merge`" connector. Finally, the first stage ends with a project operator on tokens.

Stage 2: RID-Pair Generation

The second stage, RID-Pair Generation, is expressed in lines 6-50 in Figure 5.7, excluding Stage 1. We start by scanning the DBLP dataset in line 7. Next, we tokenize each title. For each tokenized title, we order its tokens by their rank computed in the first stage. This is done in lines 12-25, by joining the set of tokens in one title with the set of ranked tokens. We introduce a compiler hint in line 23 that advises the compiler to use a broadcast join operator and to broadcast the ranked-tokens. Next we order the join results by rank, stored in the "`$i`" variable. After the tokens of each title are ordered by rank, we extract the prefix tokens in line 26. We use the "`prefix-len-jaccard`" built-in function to compute the length of the prefix for the Jaccard similarity and the 0.9 similarity threshold. The "`subset-collection`" function extracts the prefix subset of the tokens.

The same process of tokenizing, ordering the tokens, and extracting the prefix tokens is done in lines 29-42 for the CITESEERX dataset. The two streams are then joined on their prefix tokens in line 44, and we compute and verify the similarity of each of the joined pairs. The similarity is computed using the built-in "`similarity-jaccard-prefix`" function. This function is an optimized version of the "`similarity-jaccard`" function and uses the position of the common prefix token. Because a pair of records can share more than one token in their prefixes, duplicate pairs could be produced. The duplicates are eliminated by using a group-by statement in line 49.

Hyracks Plan: When this portion of the AQL query is translated to a Hyracks plan, the "`for`" loop in line 7 becomes the "`Scan 2: DBLP`" operator in Figure 5.8. The AQL compiler notices the correlation between the outer query starting at line 7 and the inner query starting at line 12. That is, the tokens of each outer record are joined at the inner level with the same set of ranked tokens. The compiler then replaces this join at the inner level with a left-outer join at the outer level, shown as the "`Hash Join (Token)`" operator in the figure, followed by a "group by" on the record id, shown as "`Preclustered Group By (RID)`" in the "`Meta 5`" operator in the figure. In the Hyracks plan, lines 8-27 become three operators: a meta operator "`Meta 3: Tokenize Join Column, Unnest`", a join operator "`Hash Join (Token)`", and another meta operator "`Meta 5: Preclustered Group By (RID), Sort (Rank), Listify, Prefix`". For the hash-join operator, one of the inputs, the ranked tokens, comes from the first stage through a split operator that replicates its input to multiple outputs. Because of the broadcast hint specified in line 23, the ranked tokens are broadcast to all the nodes using an "`M:N Replicate`" connector before the split operator. The unnested records produced by the "`Meta 3`" operator remain local to the nodes, and are sent to the join using a "`1:1`" connector. Moreover, due to the in-memory hint specified in line 20 (Stage 1), an in-memory-hash-join operator is used in the Hyracks plan. The hash-join operator loads the token ranks in memory and then streams the unnested records. Because the unnested records remain local and are also clustered by record rid, when they are grouped by rid, the compiler uses a preclustered-group-by operator without sorting the records by rid first. After the group-by operator, inside each group, the tokens are sorted by rank and are stored in a list. Notice how the compiler uses the fact the the sort is local per group and therefore does not perform a global sort.

The same transformations take place for the CITESEERX side of the join specified in lines 29-42. The two data streams are hash-partitioned by token using "`M:N Hash`"

connectors and are joined by token using a hash-join. For the joined pairs, a meta-operator, "`Meta 7: Verify RID Pair`", is used to compute the similarity and filter out the false positives. Finally, the second stage plan ends with a group-by operator that eliminates the duplicated rid pairs.

Stage 3: Record Join

The third and final stage, Record Join, is expressed in lines 1-56 in Figure 5.7, excluding Stages 1 and 2. Stage 3 consists of two joins. The first join adds the DBLP record information to each RID pair, while the second join adds the CITESEERX record information. In the first join, lines 4-53, we join the DBLP records with the RID pairs produced in the second stage. In lines 4 and 5 we loop over the two datasets, while in line 52 we express the join condition. We loop over the results of this join in line 3 and over the CITESEERX dataset in line 2. The second join is expressed in line 55. Finally, the end result is returned in line 56.

Hyracks Plan: For the Hyracks plan, this subquery is translated into two hash-joins and a projection. It is worth noting that the compiler uses the fact that the join is on rid and the input data is partitioned by rid. As a consequence, the two datasets are not repartitioned. Only the output of the second stage and the output of the first join are being repartitioned.

AQL Optimization Summary

The following list highlights some of the important optimizations that are used in the transformation from the AQL query to the Hyracks plan:

- Use of compiler hints for broadcast join, hash group-by, and in-memory opera-

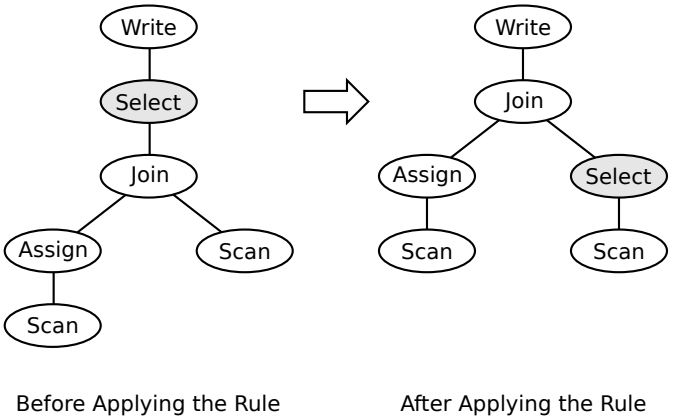


Figure 5.9: Applying the Push Select Down rule in the AQL compiler on a logical plan.

tors.

- Replacement of correlated sub-queries with left-outer join and group-by.
- Use of pre-clustered group-by without sorting the records if the group-by is on the primary key and the records are not repartitioned before the group-by.
- Use of sort as the group-by aggregator for sorting individual groups.
- Push-down of projections.

5.3.4 Fuzzy-Join Compiler Rule

Before going into the details of our fuzzy-join rule, we first need to look at how the AQL rule-based compiler works. The AQL compiler as a whole contains a set of rules (currently around 50). After a query is parsed, a logical plan is produced. This plan is provided as input to the rule manager which then starts applying the rules. For each rule, the compiler traverses the logical plan in depth-first order and tries to apply the rule to each logical operator. A given rule can be applied to a plan one time or multiple times, until a fixed point is reached, depending on the type of rule.

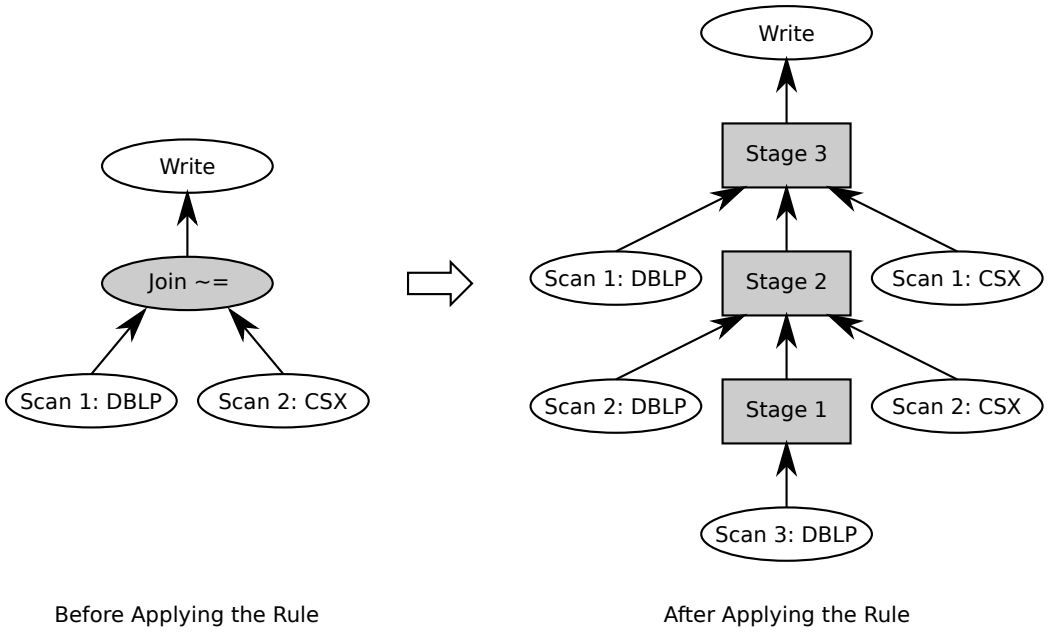


Figure 5.10: Applying the Fuzzy-Join rule in the AQL compiler on the logical plan corresponding to query FJ1.

Figure 5.9 shows how the Push Selection Down rule is applied to an example logical plan. The rule is triggered for the Select logical operator. It modifies the logical plan by moving the Select operator before the Join operator.

When a query such as example FJ1 or FJ2 is received, the compiler first detects a join. Because the join has a fuzzy predicate ("~="), the rule manager triggers the fuzzy-join rule. The goal of this rule is to rewrite a logical join operator that has a fuzzy predicate into a logical plan containing the three-stage set-similarity join algorithm. Figure 5.10 shows how the Fuzzy-Join rule is applied to the logical plan corresponding to the FJ1 query. For ease of exposition we do not show the full details of the logical sub-plans for each stage. Notice how the inputs to the original join operator are copied multiple times. The final logical plan should be similar to the logical plan corresponding to the AQL query shown in Figure 5.7.

One option to realize this rule in the compiler would have been to hard-code the three-stage logical plan into the rule and have the join operator be replaced with the

Logical Construct	AQL+ Construct
Existing Variable	Meta-variable
Existing Primary Key	Meta-variable
Existing Subplan	Meta-clause
Join	Meta-clause

Table 5.2: Logical constructs available in AQL+ and their representation in AQL+.

AQL+ Placeholder	Possible Value
TOKENIZER	none (for sets) word-tokens hashed-word-tokens counthashed-word-tokens gram-tokens hashed-gram-tokens counthashed-gram-tokens
PREFIX_LEN	prefix-len-jaccard ...
SIMILARITY	similarity-jaccard-prefix ...
THRESHOLD	$[0, 1]$

Table 5.3: Possible values for the AQL+ placeholders in Figure 5.11.

pre-constructed logical plan. This solution would make future improvements to the logical plan hard to implement, however. Moreover, writing logical plans by hand is very tedious and error prone. Instead, we choose to reuse our parser and compiler and create a new compiler-rule language very similar to AQL itself, called AQL+. The goal was to have an AQL+ query coded into the fuzzy-join rule and have it be parsed by our modified parser and then translated into a logical plan. The new logical plan would go again through all our compiler rules and benefit from all the current and future optimizations available in the compiler. Additionally, the AQL+ technology could be used in other compiler rules that require major plan changes, such as fuzzy selection.

The AQL+ syntax is a super-set of the AQL syntax. Compared to AQL, AQL+ contains two meta constructs: meta-variable and meta-clause. These two constructs

are used to express logical components in the query plan being compiled. Meta-variables are used to refer to variables from the original logical plan. For example, in the fuzzy-join rule, we use meta-variables to refer to the primary keys of the input records or the variables in the fuzzy predicate. Meta-clauses are used to refer to inputs of the AQL query and to logical constructs that cannot be directly specified in AQL, such as joins. Table 5.2 shows the list of logical constructs supported in AQL+ and the specific AQL+ construct used to represent them. The meta-constructs available in AQL+ allow us to integrate fuzzy-joins in any AQL queries with any kind of input. To accommodate for various types of data, similarity functions, and similarity thresholds, the fuzzy-join rule uses placeholders. Placeholders are parts of the AQL query which are unknown until runtime. They are used for data type or similarity-specific functions or values. Table 5.3 shows the placeholders used in our fuzzy-join rule and their possible values. The "`TOKENIZER`", "`PREFIX-LEN`", and "`SIMILARITY`" placeholders are for built-in AQL functions, while the "`THRESHOLD`" placeholder is for a numerical value.

Figure 5.11 shows the AQL+ query that is hard-coded inside the fuzzy-join rule. In the following, we describe the AQL+ subquery for each of the three stages (see comments).

Stage 1: Token Ordering. The first stage, Token Ordering, is expressed in lines 15-23 in Figure 5.11. The subquery starts with a meta-clause, "`#LEFT_3`", that denotes the left input of the original join operator that is being rewritten. Note that the same left input could be used multiple times in the rewrite rule. If there are dependencies between the reused inputs, deep copies of the inputs are created. In this case, a deep copy of the left input is created. The "`_3`" notation denotes the third such copy of this input. Next, in line 17, the "`$pk`" variable is assigned to the primary key of the records from the left input. We refer to the primary key of the left-side records using

```
1  // -- - Stage 3 - --
2  join(
3      (#RIGHT_1),
4      (join(
5          (#LEFT_1),
6          // -- - Stage 2 - --
7          (join(
8              (
9              #LEFT_2
10             let $tokensUnrankedLeft := TOKENIZER($$LEFT_2)
11             let $lenLeft := len($tokensUnrankedLeft)
12             let $tokensLeft :=
13                 for $tokenUnranked in $tokensUnrankedLeft
14                 for $tokenRanked at $i in
15                     // -- - Stage 1 - --
16                     #LEFT_3
17                     let $pk := $$LEFTPK_3
18                     for $token in TOKENIZER($$LEFT_3)
19                     /*+ hash */
20                     group by $tokenGrouped := $token with $pk
21                     /*+ inmem */
22                     order by count($pk), $tokenGrouped
23                     return $tokenGrouped
24                 where $tokenUnranked = /*+ bcast */ $tokenRanked
25                 order by $i
26                 return $i
27             for $prefixTokenLeft in subset-collection(
28                         $tokensLeft, 0, PREFIX_LEN(len($tokensLeft), THRESHOLD))
29             ),
30
31             (
32             #RIGHT_2
33             let $tokensUnrankedRight := TOKENIZER($$RIGHT_2)
34             let $lenRight := len($tokensUnrankedRight)
35             let $tokensRight :=
36                 for $token in $tokensUnrankedRight
37                 for $tokenRanked at $i in
38                     // -- - Stage 1 - --
39                     // similar subquery as "Stage 1" above
40                 where $token = /*+ bcast */ $tokenRanked
41                 order by $i
42                 return $i
43             for $prefixTokenRight in subset-collection(
44                         $tokensRight, 0, PREFIX_LEN(len($tokensRight), THRESHOLD))
45             ),
46
47             $prefixTokenLeft = $prefixTokenRight)
48         let $sim := SIMILARITY(
49                         $lenLeft, $tokensLeft, $lenRight, $tokensRight, $prefixTokenLeft, THRESHOLD)
50         where $sim >= THRESHOLD
51         /*+ hash*/
52         group by $pkLeft := $$LEFTPK_2, $pkRight := $$RIGHTPK_2 with $sim),
53         $$LEFTPK_1 = $pkLeft)),
54     $$RIGHTPK_1 = $pkRight)
```

Figure 5.11: AQL+ query used in the Fuzzy-Join Rule.

the meta-variable "$$LEFTPK_3". Again, the "_3" notation denotes the third copy of the left-side input. In line 18, we loop over the tokens of the left-side-join-attribute value. We use the meta-variable "$$LEFT_3" to refer to the left-side-join-attribute value. We also check the type of this value and if it is string, a tokenizer is used to compute the tokens. The tokenizer is introduced by the rewriting rule, before the AQL+ query in parsed, by replacing the "TOKENIZER" placeholder in line 10. The tokenizer can be specified in the original AQL query or a default one will be used. If the value of the meta-variable is already a list, then no tokenizer will be added. For the rest of the stage, the processing is exactly the same as in the hand-coded AQL query from Figure 5.7. The same precessing is done later, in lines 38-39, in the context of the second join input.

Stage 2: RID-Pair Generation. The second stage, RID-Pair Generation, is expressed in lines 6-52, excluding Stage 1. The AQL+ subquery starts with a join meta-clause, line 7. After the "join" keyword, inside the parenthesis, separated by commas, the join clause has three arguments: the left side of the join, the right side of the join, and the join condition. The left side of this rewritten join clause is expressed in lines 8-29. We start by looping over the original join's left-side input in line 9. Next, in line 10, we tokenize the left-side-join-attribute value. The tokenizer is again introduced by replacing the "TOKENIZER" placeholder. After that, the processing continues almost exactly as in the hand-coded AQL query from Figure 5.7. The only difference is in line 28, where the functions that compute the prefix length and the similarity threshold are not completely specified. Instead, the "PREFIX_LEN" and the "THRESHOLD" placeholders are used. The function that computes the prefix length depends on the similarity function. Both the similarity function and the similarity threshold could be specified in the original AQL query or defaults are used.

The right side of the join meta-clause from line 7 is expressed in lines 31-45. The

processing is very similar to that of the left side of the join clause, except that it uses the right-side input of the original join operator. Finally, the third argument of the join clause is the join condition, expressed in line 47. The rest of the processing for this stage, lines 48-52, is almost exactly the same as in the hand-coded AQL query from Figure 5.7. The only difference is that we use placeholders for the similarity function, "`SIMILARITY`", and the similarity threshold, "`THRESHOLD`".

Stage 3: Record Join. The third and final stage, Record Join, is expressed in lines 1-54, excluding Stages 1 and 2. This stage is composed of two join meta-clauses that represent the two joins required in this stage. The first join, line 4, joins the original left-side input with the output of the second stage on the left-side rid. The second join, line 2, joins the original right-side input with the output of the first join on the right-side rid.

Before the AQL+ query can be used, a few preparatory steps need to be conducted as part of the rewrite rule, such as determining the tokenizer, the similarity function, and the similarity threshold. For all three, values could be specified in the AQL query or else defaults will be used. Moreover, the required number of deep copies of the original-join-operator inputs are made and assigned to meta-variables. After these preparatory steps, the final AQL+ query is constructed by plugging-in the tokenizer and similarity parameters. Next the AQL+ query is parsed and translated to a logical plan. Finally, the original logical join is replaced with the AQL+ query's logical plan and this new logical plan is returned by the rule.

5.4 Verification Study

In this section we present a performance evaluation of set-similarity join processing in AQL. We measured absolute running time (wall clock time) as well as both speedup and scaleup. We compared the AQL implementation described in Section 5.3, as generated by the fuzzy-join rule, denoted as "AQL", with the hand-written implementation described in Section 5.2, denoted as "Native".

We ran experiments on the same 10-node IBM cluster described in Chapter 3. The Hyracks setup was the same as the one described in Section 5.2.4. We used the same two datasets described in Chapter 3, namely **DBLP** and **CITESEERX**, and we increased their sizes 2 to 10 times. We did a set-similarity join between DBLP and CITESEERX on the title using Jaccard similarity and a similarity threshold of 0.9.

To make the comparison between the AQL and Native approaches fair, we had both of them read the same input, i.e., we changed the Native approach to read the ASTERIX datasets created for the AQL approach. The input data for the AQL approach was partitioned by id, as seen in Figure 5.5(c), but the Native approach did not leverage this partitioning as this partitioning is specific to ASTERIX managed datasets. Also, during our experiments, we noticed that writing query result data to the output is currently a fairly expensive operation in ASTERIX, so we changed both approaches to not write any output. Finally, for better performance, we used closed-types for the ADM instances. Bellow is the full AQL query used in our experiments:

```
set simfunction 'jaccard';
set simthreshold '0.9';

for $dblp in dataset('DBLP')
for $csx in dataset('CSX')
```

where `$dblp.title ~= $csx.title`

return `{ 'dblp': $dblp, 'csx': $csx }`

5.4.1 Absolute Running-Time Comparison

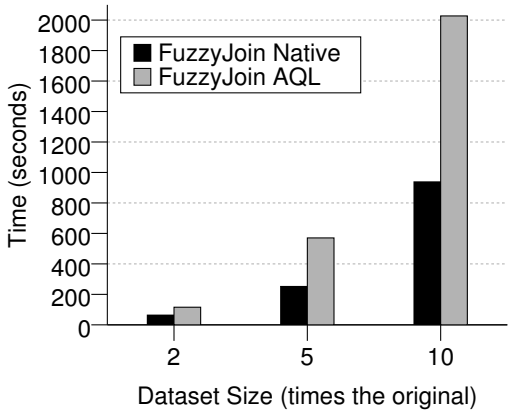


Figure 5.12: Running time for joining DBLP$\times n$ and CITESEERX$\times n$ datasets (where $n \in [2, 10]$) on the 10-node cluster.

We performed the join between the two datasets on the 10-node cluster while increasing the datasets size from $\times 2$ to $\times 10$. Figure 5.12 shows the absolute running times for Native and AQL approaches for each dataset size. We can see that there is an almost $\times 2$ performance difference between the two approaches. To account for this difference we performed an in-depth study of the two approaches. We isolated semantically equivalent sub-plans of each of the approaches and investigated where the time is spent. The reason why the AQL approach is slower than the hand-coded Native approach is three-fold:

1. In the AQL approach, each data item is currently being copied three times between ASTERIX operators and evaluators. For example, when an ASTERIX built-in function produces a data item, that data item is first copied into the function's output buffer. Then, from the function's buffer the data item is

copied into a tuple builder, which packs data into Hyracks tuples inside an Hyracks operator. Finally, the data item is written to an output frame, the unit of data transport in Hyracks.

2. As we can see by comparing the sub-plans for Stage 2 of the two approaches (Figure 5.3 for Native and Figure 5.8 for AQL), the Native approach uses a single operator for tokenizing the titles and extracting their prefix tokens, "`Tokenize and Extract Prefix`", while the AQL approach uses two complex meta operators, "`Meta 3`" and "`Meta 5`", plus a hash-join operator, "`Hash-Join (Token)`", for the same task. Moreover, a large volume of data is transferred between these three operators.

3. The use of a general data representation format in ASTERIX increases the amount of data transferred in the AQL approach by approximately 15% over the Native approach.

5.4.2 Speedup Comparison

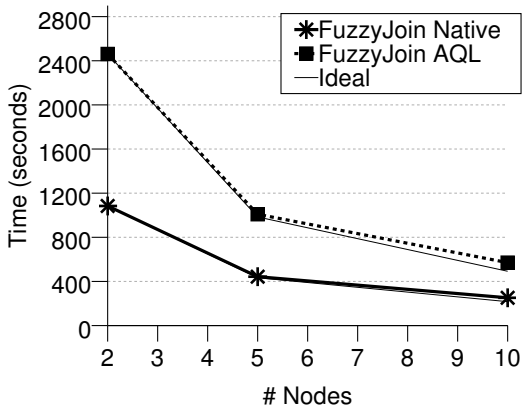


Figure 5.13: Running time for joining DBLP×5 and CITESEERX×5 datasets on different cluster sizes.

In order to evaluate the speedup of the two approaches, we fixed the dataset sizes

at ×5 their original sizes and varied the cluster size from 2 to 10 nodes. Figure 5.13 shows the running times of the two approaches as we varied the cluster size. For each approach, we also show (with a thin black line) the ideal-speedup curve relative to the running time on the 2-node cluster; that is, on the 10-node cluster, ideally, the approach should be ×5 faster than on a 2-node cluster. As we can see from the figure, the ×2 performance difference between the approaches is preserved across different cluster sizes.

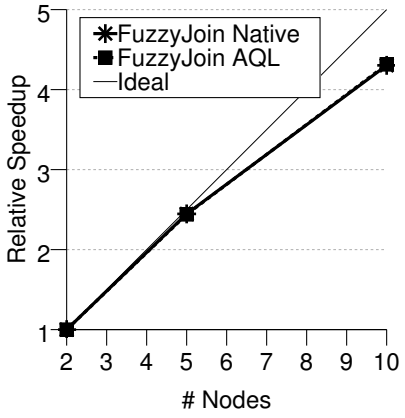


Figure 5.14: Relative running time for joining DBLP×5 and CITESEERX×5 datasets on different cluster sizes.

To better understand the speedup characteristics of the two approaches, in Figure 5.14 we plotted the same results on a relative scale. That is, the y-axis shows how much faster the running time becomes as we increase the cluster size. From this figure we can see that the two approaches have identical and almost ideal speedups. The reason for this result is that the dominant cost of our algorithm is the second stage, as we noticed before in Chapter 3. In this case, the speedup is even closer to ideal as we significantly reduced the costs for Stages 1 and 3 in Hyracks (see Section 5.2.4). Additionally, as noticed in Chapter 3, Stage 2 by itself has an almost perfect speedup.

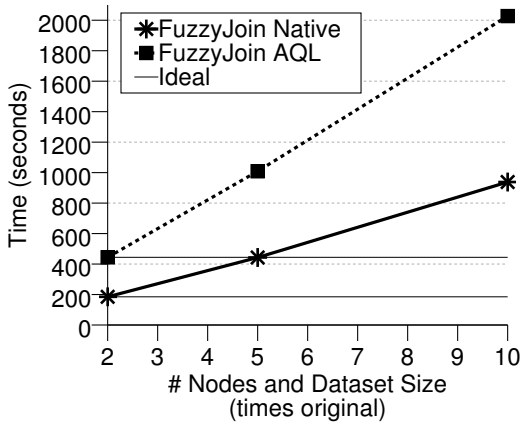


Figure 5.15: Running time for joining DBLP$\times n$ and CITESEERX$\times n$ datasets (where $n \in [2, 10]$) increased proportionally with the increase in cluster size.

5.4.3 Scaleup Comparison

In order to evaluate the scaleup of the proposed approaches, we simultaneously increased the dataset size and the cluster size by the same factor. That is, when we increased the data $\times n$ times the original size, we also increased the cluster size by a factor of n. Figure 5.15 shows the resulting running times of the approaches. Not surprisingly, the $\times 2$ performance difference between the approaches is again preserved. One the other hand, we see that the running times increase with the increase in the data size and cluster size. This performance characteristic is due to the following properties of the second stage. (1) As seen in Chapter 3, the algorithmic complexity of the algorithm used in Stage 2 is linear on the dataset size. That is, if the dataset is increased t times, each of the nodes will need t times more time to produce its result. (2) Compared to the scaleup characteristics of the approaches presented in Chapter 3, the running-time costs increase more rapidly here because we have significantly reduced the costs of Stages 1 and 3 (which had good scaleup characteristics).

5.4.4 Verification Study Summary

We summarize the study with the following observations:

- For the absolute running time, the ASTERIX AQL approach is $\times 2$ slower than the Hyracks Native approach. This difference is due in part to the generality of the AQL approach. That is, the AQL approach is expressed in a declarative language, works on various data types, and can be included in more complex queries.
- The two approaches both have almost ideal speedups due to the almost ideal speedup of the second stage of the computation.
- The scaleup of both approaches is negatively affected by the algorithmic complexity of the second stage.

5.5 Conclusions

In this chapter we designed, implemented, and verified set-similarity joins as a high-level feature in the ASTERIX query language, AQL. In the design, we focused on developing good set-similarity joins for ASTERIX's runtime platform, i.e., Hyracks. We compared the Hyracks implementation with the Hadoop implementation and observed a speedup of roughly $\times 3$ from using the Hyracks platform. Next, we focused on the integration of set-similarity joins with the high-level ASTERIX query language, AQL. We defined a syntax for set-similarity join queries in AQL, showed how the three-stage approach can actually be expressed in AQL, and then described our AQL-compiler meta-language and rewriting rule for compiling fuzzy-join queries in AQL. Finally, we experimentally verified our implementation by comparing the Hyracks

native implementation of set similarity joins with the AQL implementation. We found that the AQL implementation is slower than the Hyracks native implementation by a relatively constant factor of $\times 2$. This cost is largely due to the generality of the AQL approach: the AQL approach is expressed in a declarative language, works on various data types, and can be included in more complex queries.

Chapter 6

Conclusions and Future Work

Detecting pairs of similar records based on their string or set-column similarity is a requirement for many applications. Moreover, in today's data-driven world, it is necessary to perform such operations on multiple machines efficiently in order to handle very large data sets. In this thesis we have described and analyzed how the set-similarity join operation can be supported efficiently on two data-intensive parallel platforms, namely MapReduce and ASTERIX.

In Chapter 2 we described the foundations of set-similarity joins. We first summarized the current states of the art for single-machine set-similarity joins and parallel data-intensive platforms. In the second part of the chapter we presented a three-stage parallel set-similarity join algorithm that formed the core of this thesis.

In Chapter 3 we presented our approach to answering set-similarity joins in the popular MapReduce platform. We described a MapReduce implementation of our three-stage algorithm. For each stage, we proposed two alternatives consisting of one or two MapReduce jobs. We also presented two solutions for overcoming the out-of-memory situations that could be encountered even after the data is partitioned. We

closed this chapter with an experimental evaluation of the various approaches and our recommendations for the best approach.

In Chapter 4 we focused on a set of techniques to make the MapReduce framework adapt to runtime-conditions in order to improve the performance of fuzzy-join queries. We presented three techniques that integrate seamlessly with the MapReduce framework that help make the framework more adaptive. These techniques are general enough to improve other important classes of queries as well. We explained the performance benefits of the techniques in an experimental study involving both fuzzy-join queries and other types of queries.

In Chapter 5 we described how we have added set-similarity join support to the ASTERIX parallel, data-intensive computing platform. We first implemented fuzzy joins on the ASTERIX runtime platform, Hyracks, and conducted a performance study that showed the benefits of using Hyracks. After that, we defined a syntax for expressing fuzzy-join queries in ASTERIX's query language, AQL. We showed how our three-stage algorithm could actually be expressed in AQL, and we designed a compiler language for rewriting set-similarity join queries inside the AQL compiler. We closed this chapter with a performance comparison between the Hyracks native implementation and the AQL implementation of fuzzy-join queries.

6.1 Future Work

In this thesis we focused on set-similarity joins on large collections of records with small sets from a small domain. One possible future direction is to scale-out set-similarity joins. For this direction the following two cases could be addressed: (a) records with very large sets and (b) records with a large domain for the set elements.

For example, consider a social networking website where users express their preferences on music bands. For each band, we would have a list of users that prefer that band. A possible set-similarity join query is to find similar bands based on users' preferences. In this example, the set elements are users, and there is a very large number of unique users. Moreover, popular bands may have a very long list of users that prefer them. The algorithms in this thesis have assumed that the domain of set elements is relatively small compared with the size of the dataset and remains constant as the dataset size increases. If these assumptions are false, the processing in Stage 2 of our approach becomes challenging since the list of unique set elements is currently loaded into memory. Our algorithms have also assumed that the size of a record (as measured in their number of set elements) is small. If this assumption is not satisfied, the processing in Stage 1 and Stage 2 of our algorithms needs to be changed because we currently manipulate and send these (large) sets over the network.

Another possible direction for future work is to study parallel set-similarity joins when an index exists on the set elements of the join-column tokens. One possibility is to use an inverted list index. That is, for each set element value we could store a list of record ids whose set column containing that set element. The index could be distributed on the cluster by hashing on the set element values. If the query workload uses a fixed similarity function and a fixed similarity range, we could possibly use the prefix filter and index only the prefixes of each set. In this scenario, the first stage that accumulates the statistics would not be necessary, and the second stage might be replaced with a direct index-join operation.

Another possible direction is to study fuzzy joins when the set elements have weights associated with them. In this setting, a similarity function that incorporates the weights of the set elements would be used. The prefix filter would not be applicable

in this setting. A better way to sort the set elements in the first stage would be in decreasing order of their weights. In this way, the pairs that share high-weight elements in common might pass the similarity threshold earlier. Additionally, if a pair does not share any high-weight element in common, we could compute a similarity upper bound for the pair using the weight of the lowest-weight element not shared by the pair.

In closing, we would like to make a few general remarks. This thesis addressed the important problem of set-similarity joins which is only one type of queries in the large class of fuzzy-join queries. This thesis is only a start in this direction and there is a lot of future work to be studied. We expect this area to grow as there are many applications, including web search and social-networking websites, that require fuzzy-join queries.

Bibliography

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz. HadoopDB: an architectural hybrid of MapReduce and dbms technologies for analytical workloads. *PVLDB*, 2(1):922–933, 2009.

[2] Apache Avro. http://hadoop.apache.org/avro/.

[3] Apache Hadoop, http://hadoop.apache.org.

[4] Apache Hive, http://hadoop.apache.org/hive.

[5] Apache ZooKeeper http://hadoop.apache.org/zookeeper.

[6] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.

[7] S. Babu. Towards automatic optimization of MapReduce programs. In *SoCC*, pages 137–142, 2010.

[8] A. Balmin, 2010. personal communication.

[9] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/-PACTs: a programming model and execution framework for web-scale analytical processing. In *SoCC*, pages 119–130, 2010.

[10] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.

[11] A. Behm, V. R. Borkar, M. J. Carey, R. Grover, C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou, and V. J. Tsotras. ASTERIX: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29(3):185–216, 2011.

[12] K. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Eltabakh, C.-C. Kanne, F. Ozcan, and E. Shekita. Jaql: A scripting language for large scale semistructured data analysis. VLDB 2011.

[13] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in MapReduce. In *SIGMOD Conference*, pages 975–986, 2010.

[14] V. R. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, pages 1151–1162, 2011.

[15] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. *Computer Networks*, 29(8-13):1157–1166, 1997.

[16] Y. Bu, B. Howe, M. Balazinska, and M. Ernst. Haloop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1):285–296, 2010.

[17] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.

[18] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, page 5, 2006.

[19] J. Cieslewicz and K. A. Ross. Adaptive aggregation on chip multiprocessors. In *VLDB*, pages 339–350, 2007.

[20] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce online. In *NSDI*, pages 313–328, 2010.

[21] B. F. Cooper, E. Baldeschwieler, R. Fonseca, J. J. Kistler, P. P. S. Narayan, C. Neerdaels, T. Negrin, R. Ramakrishnan, A. Silberstein, U. Srivastava, and R. Stata. Building a cloud for yahoo! *IEEE Data Eng. Bull.*, 32(1):36–43, 2009.

[22] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[23] J. Dean and S. Ghemawat. MapReduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.

[24] J. Dean and S. Ghemawat. MapReduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.

[25] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE Trans. Knowl. Data Eng.*, 2(1):44–62, 1990.

[26] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[27] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *PDIS*, pages 280–291, 1991.

[28] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *PVLDB*, 3(1):518–529, 2010.

[29] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a highlevel dataflow system on top of MapReduce: the Pig experience. *PVLDB*, 2(2):1414–1425, 2009.

[30] GenBank, http://www.ncbi.nlm.nih.gov/Genbank.

[31] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.

[32] G. Graefe and P.-Å. Larson. B-tree indexes and CPU caches. In *ICDE*, pages 349–358, 2001.

[33] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.

[34] W. Graves. Us visit : The world's largest biometric application, March 2010. http://biometrics.nist.gov/cs_links/ibpc2010/pdfs/Graves_William_The_future_of_IDENT.pdf.

[35] M. R. Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *SIGIR*, pages 284–291, 2006.

[36] T. C. Hoad and J. Zobel. Methods for identifying versioned and plagiarized documents. *JASIST*, 54(3):203–215, 2003.

[37] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, 2010.

[38] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC Conference*, 1998.

[39] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[40] Jaql, http://www.jaql.org.

[41] Jaql 0.1. http://www.jaql.org/release/0.1/jaql-overview.html.

[42] Jaql - Fuzzy join tutorial. http://code.google.com/p/jaql/wiki/fuzzyJoinTutorial.

[43] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The performance of MapReduce: an in-depth study. *PVLDB*, 3(1):472–483, 2010.

[44] JSON. http://www.json.org/.

[45] M. Kitsuregawa and Y. Ogawa. Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the super database computer (sdc). In *VLDB*, pages 210–221, 1990.

[46] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy. A platform for scalable one-pass analytics using MapReduce. SIGMOD 2011.

[47] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.

[48] A. Metwally, D. Agrawal, and A. E. Abbadi. Detectives: detecting coalition hit inflation attacks in advertising networks streams. In *WWW*, pages 241–250, 2007.

[49] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. MRShare: sharing across multiple queries in MapReduce. *PVLDB*, 3(1):494–505, 2010.

[50] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic optimization of parallel dataflow programs. In *USENIX Annual Technical Conference*, pages 267–273, 2008.

[51] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD Conference*, pages 1099–1110, 2008.

[52] O. O'Malley and A. C. Murthy. Winning a 60 second dash with a yellow elephant. Technical report, Yahoo! Inc., 2009.

[53] 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 Conference*, pages 165–178, 2009.

[54] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. WCB/McGraw-Hill, 2002.

[55] M. Sahami and T. D. Heilman. A web-based kernel function for measuring the similarity of short text snippets. In *WWW*, pages 377–386, 2006.

[56] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD Conference*, pages 743–754, 2004.

[57] Skewed join in Pig. http://wiki.apache.org/pig/PigSkewedJoinSpec.

[58] E. Spertus, M. Sahami, and O. Buyukkokten. Evaluating similarity measures: a large-scale study in the Orkut social network. In *KDD*, pages 678–684, 2005.

[59] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Z. 0002, S. Anthony, H. Liu, and R. Murthy. Hive - A petabyte scale data warehouse using Hadoop. In *ICDE*, pages 996–1005, 2010.

[60] Web 1T 5-gram Version 1, http://www.ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=LDC2006T13.

[61] J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K.-L. Wu, and A. Balmin. FLEX: a slot allocation scheduling optimizer for MapReduce workloads. In *Middleware*, page to appear, 2010.

[62] C. Xiao, W. W. 0011, X. Lin, and H. Shang. Top-k set similarity joins. In *ICDE*, pages 916–927, 2009.

[63] C. Xiao, W. Wang, and X. Lin. Ed-join: An efficient algorithm for similarity joins with edit distance constraints. In *VLDB*, 2008.

[64] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140, 2008.

[65] XQuery 1.0: An XML query language. http://www.w3.org/TR/xquery/.

[66] H. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker Jr. Map-Reduce-Merge: simplified relational data processing on large clusters. In *SIGMOD Conference*, pages 1029–1040, 2007.

[67] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, pages 1–14, 2008.

[68] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, pages 265–278, 2010.

[69] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *HotCloud'10: Proceedings of the 2nd USENIX workshop on Hot topics in cloud computing*, 2010.

[70] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, pages 29–42, 2008.

[71] J. Zhou, P.-Å. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *ICDE*, pages 1060–1071, 2010.