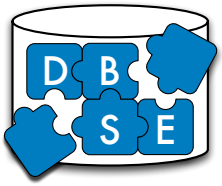
Otto-von-Guericke-Universität Magdeburg
Faculty of Computer Science



Databases
and
Software
Engineering

Master Thesis

An Evaluation of Deep Hashing for High-Dimensional Similarity Search on Embedded Data

Author:

Rutuja Shivraj Pawar

October 02, 2019

Internal Supervisors:

Prof. Dr. rer. nat. habil. Gunter Saake, M.Sc. Gabriel Campero Durand
University of Magdeburg

External Supervisor:

Prof. Dr.-Ing. Sebastian Michel
Technische Universität Kaiserslautern

Pawar, Rutuja Shivraj:
An Evaluation of Deep Hashing for High-Dimensional Similarity Search on Embedded Data
Master Thesis, Otto-von-Guericke-Universität Magdeburg
Faculty of Computer Science, 2019.

Abstract

In today's era, the rate at which data is accumulating is exponential, which makes it increasingly challenging to retrieve relevant information. In such a scenario, high-dimensional similarity search serves as a popular method to extract relevant information from large data volumes or Big Data, and it further drives different Machine Learning (ML) tasks including, Near Duplicate Detection & Location Recognition. However, Big Data, due to its characteristics, poses a variety of challenges to ML applications, such as high class imbalance, the need for feature engineering to support heterogeneous data and the need for efficient solutions for queries over array data. Consequently, in this thesis, we aim to optimize the data analytics pipeline for the utilization and effective management of feature engineering data (embedding data), offering as one of the solutions in the context of high-dimensional similarity search. In doing so, we evaluate the impact of similarity-preserving hashing on helping with data blocking and skipping for ML applications of supervised entity resolution and top-k similarity search.

Precisely, we make the following contributions:

First, we utilize and work with embedding data, as an approach to highlight semantic similarity in the data, thus making it more manageable. In doing so, we experiment with three dataset pairs from two different domains, Bibliographic and E-commerce, with their attributes embedded using a fastText pre-trained model. Further, based on its fast query speed and low memory costs, we consider similarity-preserving hashing as the technique to manage these embedding data and efficiently support high-dimensional similarity search. Specifically, we consider two hashing techniques, Locality Sensitive Hashing (LSH) being data-independent, and Learning To Hash (L2H) being data-dependent.

Second, based on well-defined metrics, we experimentally evaluate the efficiency and classification accuracy of LSH - Super-Bit, with a focus on the task of supervised entity resolution.

Third, based on the same metrics, we experimentally evaluate and compare LSH - Super-Bit with L2H - Deep Hashing. In doing so, we utilize our designed *Deep Hash Neural Net (DHNN)*, based on the literature. This designed network serves as our main contribution in offering a deep hashing neural network generalized to work with embedding data. In this evaluation, we are able to report a superior performance of L2H - Deep Hashing over LSH - Super-Bit, for the task of supporting supervised entity resolution.

Finally, based on the outcome of the experimental evaluation, we further evaluate the runtime performance and speed-up brought by L2H - Deep Hashing to top-k similarity search queries in Apache Spark, using different file formats.

Acknowledgement

Through writing this acknowledgement, I wish to express my deep sense of gratitude to all the people involved in the realization of this Master Thesis.

First and foremost, I would like to express my sincere gratitude to my supervisor and mentor M.Sc. Gabriel Campero Durand, for his continuous guidance, motivation, and expert advice throughout this thesis research. The enriching discussions that we had combined with his constructive feedback and immense knowledge added to a valuable contribution to this research. I could not have imagined having a better mentor than him for my thesis.

I am grateful to Prof. Dr. rer. nat. habil. Gunter Saake for giving me the opportunity to conduct my thesis research under his chair and sharing his valuable feedback as the first reviewer for this thesis. The excellent research environment and the valuable ongoing research at his Database and Software Engineering Workgroup always motivated me to write my thesis under his chair. I am also thankful to Prof. Dr.-Ing. Sebastian Michel, with whom I had the opportunity to interact and have an insightful discussion after his Keynote speech on *Similarity Search and Data Exploration over Entity Rankings* at the 31st GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken) in Saarburg. I would like to especially appreciate his quick response in acceptance as the second reviewer for this thesis and sharing his valuable feedback.

I would also like to thank M.Sc. Xiao Chen for her experienced guidance in the setup and experiment with the Apache Spark cluster. Additionally, I would like to thank Technical Administrator, Dipl.-Inf. Steffen Thorhauer for his quick response helpful in the timely setup of the Spark cluster.

Finally and most importantly, I would like to thank my family: my grandparents *Ajji* and *Ajoba*, my parents *Mamma* and *Pappa*, and my little sister *Amarja* for their immense continuous support, motivation and good wishes in all the ventures of my life that I undertake!

Declaration of Academic Integrity

I hereby declare that this thesis is solely my own work and I have cited all the external sources used.
Magdeburg, October 06, 2019

_____

Rutuja Shivraj Pawar
Matriculation No. 220051

Contents

List of Figures

List of Tables

1. Introduction

1.1 Motivation

Applications drive the progress of data management technologies. Nowadays, Machine Learning (ML) programs are one key class of such applications. They are becoming increasingly relevant since they facilitate the process of extracting meaningful information from ever-growing amounts of business data. In today's era of big data, realizing successful ML applications has become increasingly challenging [1]. However, research is showing that some of these challenges can be addressed by efficient data management and processing. Figure 1.1 depicts these different challenges in the context of the four Big Data processing characteristics. Some relevant challenges include, *class imbalance* (i.e., real-world data can exhibit very few examples of a particular class in a classification task, requiring special handling), the requirement for complex *feature engineering* (i.e., extracting appropriate data representations for a particular task), *data heterogeneity* (i.e., it is difficult to have machine learning algorithms capable of working well over dimensions that are highly diverse in data types), the requirement for *efficient processing* of enormous amounts of data (i.e., the need for fast large-scale processing frameworks), among others.

To overcome these challenges, there are two main solution categories, firstly, *data, processing and algorithm manipulations to handle big data* and secondly, the *creation and adaption of different ML paradigms with existing algorithm modifications.* In the context of the first category, the respective three types of manipulations can take place in the different stages of the data analytics pipeline, from *data extraction* to *decision making* [14], as depicted in Figure 1.2. It is in this category, where data management solutions are called upon to assist companies in their ML applications.

In recent years, embeddings have become popular as a first step *data manipulation* technique in data analytics pipelines. This technique is helpful to alleviate the curse of dimensionality and to standardize feature engineering in domains where data is non-euclidean (e.g., images, text, graphs)[1]. Embedding is the process of mapping entities of such kind of data, into

individual dense continuous vectors, with a given number of dimensions. It can also be defined as a mapping of discrete variables (e.g., words) to a vector of continuous numbers. Different embedding processes are also able to highlight latent similarity in the data, through the placement of similar inputs close together in the embedding space [15]. As a result of the embedding process for each entity in a dataset, a number of dense numerical vectors is produced which can be used as features for ML. The embedded data can also be used for similarity retrieval tasks, complementing the search over raw data. However, loading and managing a large amount of embeddings for their processing in productive applications can be a problem since most data management solutions are not tailored to such kind of vector data nor to the kind of operations done with them. Hence, effective storage and retrieval mechanisms for these embeddings are likely to play an important role moving forward.

In this thesis, we aim to study some optimizations of the data analytics pipeline (Figure 1.2) for the utilization and effective management of embedding data. Specifically, in the context of the *processing manipulation* techniques which focus on modifying how data is processed and stored to improve ML performance, *similarity-preserving hashing* is one technique which has become effectively popular in managing vector data, due to its fast query speed and low memory costs for supporting top-k similarity search [16–24]. However, most research in this domain has either focused on the application side or has been scoped for multimedia data, without considering general embeddings. As a result, practitioners lack published experimental evaluations that could help them in selecting a technique to manage embeddings in large-scale processing, especially concerning embeddings for more traditionally structured textual data. In addition, practitioners lack sufficient information on the expected performance gains from selecting a given similarity-preserving hashing technique.

In this thesis, we experimentally study and evaluate two hashing techniques, that can be applied to efficiently manage embedding data, namely Locality Sensitive Hashing (LSH) and Learning To Hash (L2H). Due to the lack of availability in general solutions for L2H, we develop our own approach to supervised hashing with deep neural networks, known as *Deep Hash Neural Net (DHNN)*. We study the hashing techniques, considering how they can be applied for entity resolution (i.e., contributing to the blocking process), and for top-k similarity search (i.e., contributing to data skipping). Some key components of our study are the use of datasets that are standard for entity resolution, and that represent structured textual data. We also use a standard pre-trained embedding model, FastText [25], and we use Apache Spark[2]studying different Hadoop file formats. Consequently, with our work, we seek to realistically evaluate the impact of similarity-preserving hashing on helping with data blocking and skipping for ML applications of supervised entity resolution and top-k similarity search.

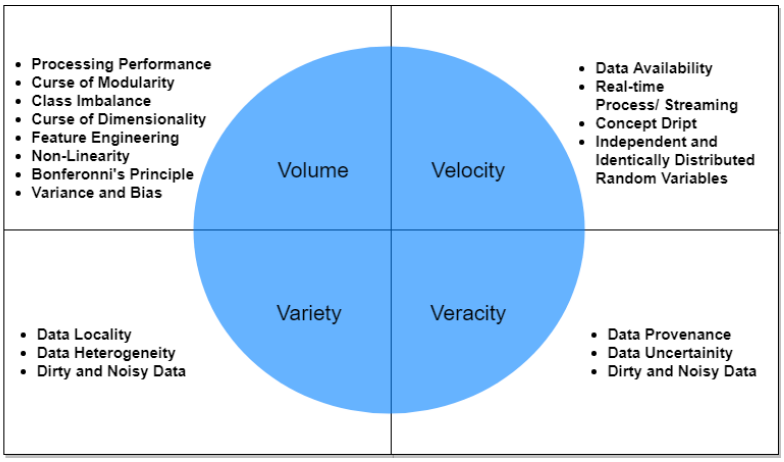


Figure 1.1: Machine Learning Challenges Associated with Big Data Characteristics (adapted from L'heureux et al. [1])

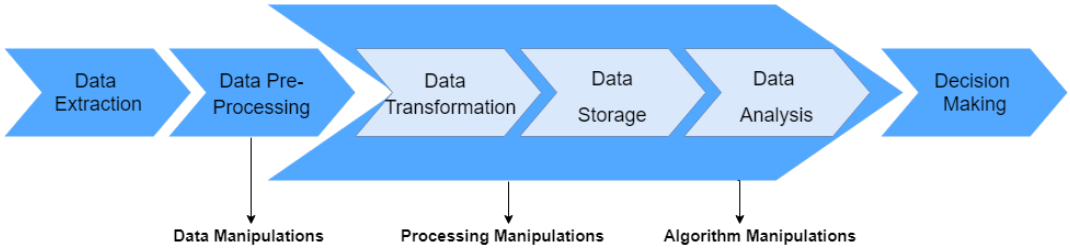


Figure 1.2: Data Analytics Pipeline (adapted from L'heureux et al. [1])

1.2 Previous Work

On viewing embeddings as a form of vector array data and techniques used for array data management, previous work related to the area of research of this thesis includes the research of Sun et al.[26], which highlights the idea of data skipping to scan a large amount of data quickly. The underlying mechanism of data skipping lies in the fact that the data is organized into blocks with metadata maintained for each of the block, which then can be utilized by the query to skip irrelevant blocks of data. Sun et al.[26] propose a fine-grained data layout framework called Generalized Skipping-Oriented Partitioning and Replication (GSOP-R) which maximizes query performance through aggressive data skipping. The GSOP-R works on a specific set of features summarizing the workload patterns. It then uses these features to transform the incoming data into a small set of feature vectors performing clustering on them for blocking, with the flexibility of scope for both horizontal and vertical partitioning schemes. GSOP-R is also further prototyped and evaluated on Apache Spark with the partitioned data being stored into Apache Parquet[3], a columnar storage format.

Further, Stonebraker et al.[27] provide practical insights, by considering how to build data systems to support the array data model. This would enable effective data processing in

the context of scientific data [28], which can be best modeled as an array. In a distributed cluster setup, authors consider a division of the array data into storage chunks and study techniques for its effective partitioning, with the goal of maintaining the storage load evenly among the nodes. Consequently, authors report on a hash partitioning scheme to allow for fine-grained storage partitioning. Additionally, authors also highlight the important role of k-nearest neighbors, as an application that needs to be easily and naturally executed on an array data model. Stonebraker et al.[27] precisely develop SciDB [29] as a new open-source scientific database, built on the array data model to support multi-dimensional arrays with any number of dimensions.

1.3 Initial Research Questions

Below are the initial research questions formulated to be answered through this thesis:

- **RQ**$_1$: What is the best possible coverage and block distribution achievable using a standard high-dimensional Locality Sensitive Hashing technique?
- **RQ**$_2$: How does a Learning To Hash technique compare to the standard high-dimensional hashing technique, considering its coverage and block distribution, and how it contributes to the blocking task of supervised entity resolution?
- **RQ**$_3$: What is the performance benefit achievable by hashing with this technique, in contrast to the other approach, for supporting top-k similarity search in Spark?

1.4 Research Methodology

The selection of a proper research methodology is a crucial task before starting the research, as it provides a systematic approach to solve a problem. This is highly influenced by the field in which the research is being carried out since different research methodologies cater to different aspects of a domain. Considering that our area of research is a combination of Data Science and Big Data, we required a research methodology which is well-suited for this area. As compared to the different Data Science process models (e.g., CRISP-DM), Data Science Edge (DSE) [2] is an enhanced process model which well accommodates both data science activities and Big Data technologies. Thus, DSE was selected to be an appropriate research methodology to carry out structured research for this thesis.

As shown in the Figure 1.3, DSE consists of five iterative and adaptive process stages, which can be repeated to incrementally improve the obtained result, as described below:

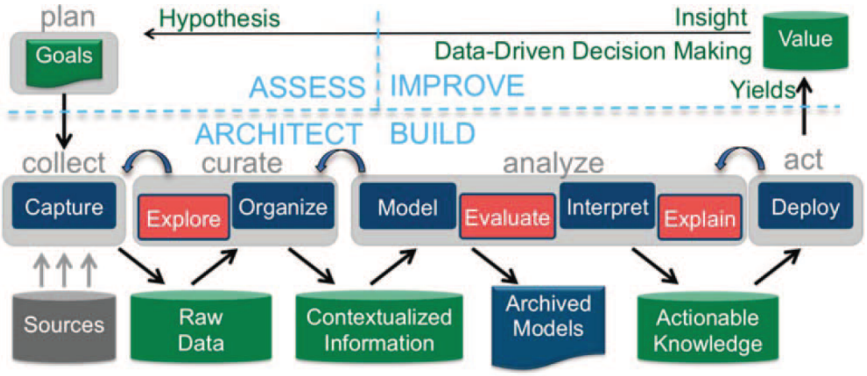


Fig. 4. Data Science Edge phases, activities, and data maturity

Figure 1.3: Different phases of Data Science Edge [2]

(i) **Plan** This phase deals with the planning and a rough estimation of the activities to be carried out, with a clear view of the objective to be achieved. Understanding and analyzing the requirements with the identification of the critical areas are also a part of the Plan phase.

(ii) **Collect** This phase deals with the collection of the required data for the project and its management. Understanding the sources and the nature of the data to be collected in order to devise appropriate data management strategies are the activities that constitute the Collect phase.

(iii) **Curate** This phase deals with visualizing the data in order to understand its different characteristics. A set of tailored data cleaning activities to remove potential outliers or errors from the data, that would otherwise adversely affect the analysis on the data, constitute the core tasks of the Curate phase.

(iv) **Analyze** This phase deals with analyzing the carefully curated data to obtain meaningful insights from it. Devising, modeling, and implementing technical strategies and solutions to obtain useful results from the data to answer the given problem are the tasks of the Analyze phase.

(v) **Act** This phase deals with communicating and presenting the obtained results from the Analyze phase in a simplified and clear way, mainly through meaningful visualizations. Description of the use of the obtained results and its intuitive presentation are the tasks that conform to the Act phase.

Consequently, a complete pass of these phases was carried out separately for each of the research questions in this thesis work.

1.5 Thesis Structure

This remainder of this thesis is structured as follows:

- Chapter 2 provides an understanding of the background and related work in the research area of this thesis.
- Chapter 3 provides an overview of the thesis design, its prototypical implementation details, and the experimental setup.
- Chapter 4 presents the experimental evaluation results of the Locality Sensitive Hashing (LSH) technique, encompassing the block distribution, and supervised entity resolution.
- Chapter 5 presents the experimental evaluation results of the Learning To Hash (L2H) technique, encompassing the block distribution, and supervised entity resolution.
- Chapter 6 presents the experimental evaluation of the runtime performance achieved when applying the Learning To Hash (L2H) technique on similarity search queries in Apache Spark.
- Chapter 7 presents the related work for this thesis.
- Chapter 8 concludes the thesis research work by summarizing some important findings, and elaborating on possible future work for the research area of this thesis.

[1] As an illustration for how mainstream embeddings have become, Microsoft has released entity embeddings for all authors of their open-source Microsoft Academic Graph http://ma-graph.org/entity-embeddings/, aiming to help practitioners of scholarly network analysis.

[2] https://spark.apache.org/

[3] https://parquet:apache:org/

2. Background

This chapter focuses on understanding the background in the research area of this thesis. It covers the important relevant literature with a detailed description to create a solid foundational understanding of the research area. It aims to have a critical evaluation of the literature in the research area of this thesis, eventually contributing towards the design and implementation of the specific research topic for this thesis.

This chapter is structured as follows:

- Section 2.1 presents a brief description of the steps performed in the literature search process to identify the important literature on the research topic.
- Section 2.2 presents an overview of the idea of embeddings and their applicability.
- Section 2.3 presents different aspects related to large-scale data processing, a domain where solutions to manage embedding data, supporting tasks like efficient high-dimensional similarity search, are not available at the moment.
- Section 2.4 presents a brief overview of machine learning applications on large scale processing, in the context of embedding data.
- Section 2.5 presents the different optimizations techniques applicable for managing embeddings as a form of array data. Here we also describe the idea of high-dimensional hashing on embedding data with a focus on two hashing techniques, namely Locality Sensitive Hashing (LSH) and Learning To Hash (L2H).
- Section 2.6 presents a summary of the content discussed in this chapter.

2.1 Systematic Literature Review

2.1.1 Literature Search Process

Our Literature Search process was carried out as follows:

(i) **Define a list of keywords on your subject area**

The search was carried out on five different databases namely, IEEE Xplore, Springer Link, Google Scholar, Elsevier and ACM Digital Library selected based on their high quality of published papers and for a non-biased search. Additionally, as the research topic encompasses various areas, the focus was given on prominent areas to have a scoped search. The three main areas included Large-Scale Data Processing, Machine Learning and Data Management Interfaces in the Context of Embeddings and Optimizations for Embeddings and Array data.

The search terms for each of the identified areas were devised as follows,

- Identify an initial list of keywords for the selected area
- Generate a combination of keywords which will output good results
- Run a search query on the database. Refine the initial keyword list based upon the inspection of the retrieved papers, and then generate new combinations of keywords as well.

The search queries based on the above search terms were then executed on the selected databases. The output results of the search were then further reduced by the inclusion and exclusion criteria. The citation lists of the shortlisted papers were also used to identify more papers, avoiding papers by the same authors.

Sample Search Queries	**Total Hit**
“large-scale processing” AND “survey” AND “hadoop” AND “mapreduce”	307
“apache spark” AND (“word embedding” OR “benchmark” OR (“mllib” AND “applications”))	4700
“spark optimizer” OR ((“Catalyst” OR “Tungsten”) AND “apache spark”)	439
“apache parquet” OR (“avro” AND “orc”) OR “apache kudu” OR “apache arrow”	457
“in-memory grid” OR “in memory grid” OR “apache ignite” OR “alluxio”	664
“embedding” AND “survey” AND “fasttext” AND “word2vec”	283
“SciDB” AND “hashing”	142

Table 2.1: Literature Search Queries

(ii) **Select among those papers**

The paper set was refined based on the following inclusion and exclusion criteria:

Inclusion criteria: A paper must be either published in a book, journal or conference proceedings, It must be relevant to the topic under consideration, It should give a large-scale view of the field, and It must be written in English.

Exclusion criteria: A paper which does not provide relevant insights on the selected area, Is too specialized for the selected area or describes a highly specialized application for the selected area, Is a book review, Is a thesis or Is not peer-reviewed.

(iii) **Compare the papers you select with your original paper (called "seed paper" hereafter):**

The papers *"Big Data 2.0 Processing Systems A Survey"* [30] and *"Text Similarity in Vector Space Models: A Comparative Study"* [15] and *"SciDB DBMS Research at M.I.T"* [29] were selected as the seed papers for the main areas on Large-Scale Data Processing, Machine Learning and Data Management Interfaces in the Context of Embeddings and Optimizations for Embeddings and Array data respectively, based on its relevancy to the area and matching keyword search and citations. The selected papers based on the inclusion and exclusion criteria were then compared with the respective seed papers, and the paper set was then further refined.

(iv) **Use the seed paper to expand the list of papers from 1.2**

Based on the seed papers, the set of papers was further expanded and then refined, eliminating irrelevant material.

A total set of 97 main scientific papers was then considered for the research. Consequently, during the course of this entire research, based on these main papers, this set builds upon and expands into a total of 235 scientific papers constituting the research literature.

2.1.2 Overview of Selected Papers

Table 2.2 depicts an overview of the important scientific literature for the respective areas.

2.2 Overview on Embeddings

An embedding is the translation of high-dimensional vectors or non-euclidean data, into a low-dimensional space[1]. In the context of textual input, word embeddings are able to transform each work in text into an individual dense continuous vector, with a given number of dimensions, while preserving their semantic information through the placement of similar inputs close together in the embedding space [15]. Text embedding methods can be categorized into, *Count-based* which rely on a bag-of-words model where the order of the words is ignored eg., Topic Models and *Prediction-based* based on sequence-of-words models where the order of the words is taken into account eg., Neural Models. Additionally, embeddings can also be generated for input data formats including graphs, audio/video etc. As seen in

Category	Literature
Overview on Embeddings	[1] [31] [32] [33] [34] [35] [36]
Large-Scale Data Processing	[30] [37] [38]
Examples of Large-Scale Data Processing Systems	[39] [40] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [3] [55] [56] [57] [58] [59] [60]
Optimizers for Dataflow Large-Scale Data Processing	[5] [61] [62] [4] [63] [64] [65]
Storage for Large-Scale Data Processing	[66] [67] [68] [69] [70] [71] [72] [6] [73] [74] [75] [76] [77] [78] [79] [80]
Benchmarks	[81] [82] [83] [84] [85] [86] [87] [88]
Machine Learning and Data Management Interfaces in the Context of Embeddings	[15] [89] [90] [91] [92] [93] [94] [95] [96] [97] [98] [7]
Optimizations for Embeddings and Array Data	[29] [27] [99] [100] [101] [102] [103] [104] [105]
Locality Sensitive Hashing	[106] [107] [8]
Learning To Hash	[108] [18] [9] [11] [109] [110] [10]
Supervised Entity Resolution	[12] [13]

Table 2.2: Overview of Important Scientific Literature

Figure 2.1, as a visualization for real embeddings, it depicts geometrical relationships that capture semantic relations as between country and its capital on the *Left* and verb tenses on the *Right*. Consequently, embeddings make it easier and efficient to do Machine Learning (ML) on relatively large inputs with the facility of learning and reusing the embeddings as features across models, for different tasks. The embedding space also constitutes a sort of meaningful space which gives opportunities for the ML system to detect semantic patterns useful for the learning tasks. In addition, the embedding space, with its enhancing of similarities, can help in similarity search tasks, which are common in information retrieval or recommender systems.

In recent years many approaches to embeddings for different datasets, with practitioners making available pre-trained models such as Glove, Word2Vec and fastText. Surveying this variety of models is beyond the scope of this thesis, however, we refer the interested reader to authoritative surveys on the topic, concerning textual [111] and network-structured data [112].

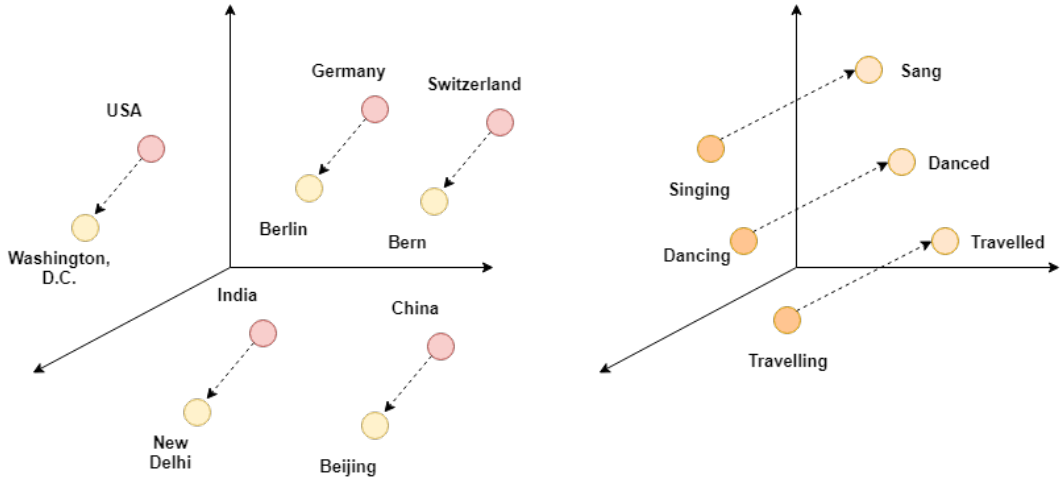


Figure 2.1: Visualization of Embedding Analogies of Country-Capital on the *Left* and Verb Tense on the *Right*

Considering the area of applicability and influence of embeddings specifically for ML, Figure 1.1 depicts on different challenges faced by ML applications in context of the four Big Data characteristics namely, *Volume*, *Velocity*, *Variety* and *Veracity*. In such a scenario, embeddings act as a data manipulation technique in the data analytics pipeline [14] which helps to alleviate the curse of dimensionality and increase data manageability through preserving its semantic relationships.

To illustrate the applications of embeddings in ML, we observe the work of Pablos et al. [34], where word embeddings are employed in machine learning based sentiment analysis, providing input features to complex supervised classification systems, thus obtaining sentiment classifiers. Regarding applications of embeddings in recommender systems, we can consider the work of Guerraoui et al. [36], who present DEEPCIP a word embedding-based recommender inspired from the neural word embeddings approach Word2vec [113, 114], designed to select and recommend items to a user based on their implicit feedback measuring their item consumption through time, with competitive performance. Further, Kose et al. [31] propose a song recommender system which utilizes Word2vec to generate a vector representation of the songs. Word2vec is further adapted to the Apache Spark[2] Big Data framework, to generate a user-specific playlist when executed on a distributed vector representation of songs.

Concerning the practical aspect of working with embeddings in modern data management systems, Svyatkovskiy et al. [32] evaluate the Spark framework for a data-intensive ML problem and demonstrate its efficiency. As embeddings are a form of multi-dimensional vector array data, related state-of-art works in this area, design and implement a hierarchical index strategy for Spark with HDFS to efficiently query and process geospatial raster data [33], propose a scalable approach towards efficiently analysing large Earth observation (EO) datasets through the representation of collection of EO's into multi-dimensional arrays coupled with the utilization of SciDB [27, 29] for computationally intensive analytics [35].

In the next section, we provide a comprehensive overview, to give a broad contextual understanding of large-scale data processing solutions. This is done in consideration that such kind of solutions seem to require novel methods for managing embedding data, since this kind of data has not been the focus of these solutions so far.

2.3 Large-Scale Data Processing

In today's digital era, we are overwhelmed with the tremendous and explosive growth of data from various sources and in different formats, also termed as *Big Data* [115]. Consequently, Big Data poses different challenges for its underlying frameworks in its large-scale storage and processing, which has led to constant evolution in the large-scale data processing systems [30]. Big Data is not useful in and of itself unless a utility is associated with it. Large-scale processing of such data to extract meaning and value out of it has, therefore, has become increasingly important to render utility to this data, thus serving as a foundation to take data-driven decisions visualizing a complete picture of the concerned application domain. Various academic and industrial sectors are increasingly dependent upon the knowledge extracted from Big Data, establishing it as a key resource in today's modern world, which further makes Big Data tools crucial for the optimized and efficient processing of this large-scale data. Large-scale data processing systems are almost synonymous with Big Data tools. They can be categorized into different system layers based on their features and applicability, namely, *Data Storage Layer, Data Processing Layer, Data Querying Layer, Data Access Layer and Management Layer* [37]. Consequently, the increased demand for large-scale data processing and analysis, caused an increase in the development of large-scale data processing systems from both the academia and industry [38].

In this section, we present in detail the different aspects of large-scale data processing with the description of available frameworks which facilitate large-scale data processing applicable for different application scenarios, the optimizations for large-scale data processing, the different available optimized storage possibilities and finally we also outline the benchmarking evaluations which help to quantify and evaluate the performance of the different large-scale data processing frameworks. The goal of this section is to provide a comprehensive context for our current and follow-up research, informing our choice of platform to select for studying the use of embeddings.

2.3.1 Examples of Large-Scale Data Processing Systems

There exist several Large-Scale Data Processing Systems designed to provide different functionalities. Below presented are some of the examples of Large-Scale Data Processing Systems accordingly categorized:

(i) **Bulk-Synchronous Parallel (BSP)** The Bulk-Synchronous Parallel [43] model acts as a bridge between the theoretical and practical aspects for the implementation of parallel computation. BSP is neither a programming or hardware model, but lies in between them acting as a standard for mapping high-level programs to machines, maximizing

the efficiency for parallel computation. The BSP model compromises of three attributes, namely, a number of *components* which perform memory/processing functions, a router which delivers messages between the components, and a synchronization mechanism between the components. Computation in a BSP model consists of a sequence of supersteps, during which each processing component performs a local computation with the sending and receiving of messages to and from other components. In the processing at each superstep, the local computation depends only on the local data present in-memory at the start of each superstep with a communication pattern known as *h-relation*, meaning that each processing component can send and receive a maximum of defined h-messages. As the components and router are different entities, this creates a simplified model which separates the tasks of computation and communication. To ensure performance guarantees, the BSP model utilizes a *Barrier synchronization* enforcing all the processing components to synchronize at a specific point, the end of a superstep. Additionally, to reduce waiting times, the synchronization mechanism can be switched off for a subset of components, leading to certain asynchronous behavior. Consequently, the BSP model aims at achieving guaranteed performance at a near to optimal utilization of distributed processors, facilitating high-throughput in parallel computations. As an example application, the high-level organization of Pregel [42] (introduced later in ix Large-Scale Graph Processing) programs are based on the BSP model.

(ii) **MapReduce** MapReduce [44] is a computation framework for the efficient processing of large datasets in a parallel distributed environment. Unlike BSP, it is based on a simplified restricted programming model which automatically parallelizes the computations providing effective fault-tolerance. In the MapReduce programming model, users can express the computation by implementing two functions, namely, the *Map* and the *Reduce*. Given a set of input key/value pairs, the Map function processes the input to produce intermediate key/value pairs as an input to the Reduce function which then merges all the values associated with the same key. In a parallel distributed execution environment, the input data to be processed is automatically partitioned into M splits by the MapReduce library with the workload distributed among a set of worker nodes which perform the Map and the Reduce tasks and are all controlled by a master. The processing output of MapReduce is then available as a set of output files corresponding to each Reduce task. Additionally, MapReduce implements a number of optimizations to minimize network traffic, including locality-aware scheduling. Consequently, MapReduce as a restricted programming model abstracts the complexities related to large-scale parallel distributed processing (e.g., load-balancing, fault-tolerance), thus providing ease of use towards expressing a large variety of problems.

(iii) **Dataflow** Computation frameworks for batch-oriented dataflow, like MapReduce do not efficiently cater to *data reuse* of intermediate results across multiple computations. As a result, the utilization of Resilient Distributed Datasets (RDDs), as proposed by Zaharia et al. [45], or similar concepts, which store data in-memory and can recover it without replication through tracking the lineage graph consisting of operations used to build the data, becomes a more optimal alternative. Dataflow engines are this

framework that use such computational graph, without the requirements of batch processing and synchronization points, as in Map Reduce. Spark, with its internal use of RDDs, is an example of a dataflow engine. In this example, each RDD is a read-only, partitioned collection of records with immutable nature and can be created through deterministic operations on stable storage data or from other RDDs. Users can control RDD aspects, namely, partitioning and persistence allowing for specification for storage reuse and optimize placement strategies. RDDs implemented in Spark[2]thus allow users to cache previous computations in-memory making them well-suited towards iterative machine learning applications and interactive data mining. Spark provides a functional programming API in the Scala[3]programming language to manipulate RDDs as distributed memory abstractions with coarse-grained transformations like map, filter, and join, and additionally also provides support for interactive query of big datasets from the Scala interpreter. Consequently, Spark with RDD can efficiently express a number of cluster programming models, namely Pregel [42], HaLoop [116] etc. and emerges as an effective programming model for batch analytics. However, Spark has over 150 configurable parameters, which makes it a challenging task to find the optimal parameter configuration for applications and clusters in order to reduce the processing time and enhance the overall performance. Considering parameter tuning, experimental studies [46] provide a systematic approach for parameter setting through investigating the impact of some of the important tunable parameters related to shuffling, compression and serialization on the performance of Spark applications.

(iv) **Timely Dataflow** Timely Dataflow is a model of data-parallel computation which extends the traditional dataflow model through associating each communication event with a virtual timestamp. As an example, Naiad [41] presents itself as a single platform for executing stateful data-parallel cyclic dataflow applications in a distributed setup with high throughput and low latency. Naiad extends traditional dataflow to provide support for incremental and iterative data-parallel computation. The virtual timestamps mechanism constituting the timely dataflow in Naiad facilitates for efficient and lightweight co-ordination in a distributed setup. These logical timestamps are designed based on a restricted looping structure present in the timely dataflow graphs in which vertices of the directed graph are organized into possible nested loop contexts containing the cycles in the graph and associated with three special nodes namely ingress, egress and feedback. The edges entering a loop context must pass through the ingress node and the ones leaving must pass through the egress node. Additionally, at least one feedback node is present in a loop context. These vertices then act upon and adjust accordingly the timestamps of the messages passing through them. Additionally, Naiad provides optimization techniques to reduce the communication overhead in a distributed setup and also for fault tolerance mechanism providing consistent recovery in case of failures. Naiad offers a flexible structure in a way that many powerful programming models can be built upon its low-level primitives, thus enabling diverse tasks like streaming data analysis, iterative machine learning and interactive graph mining.

(v) **Streaming** Streaming relates to the handling and processing of data continuously received from different sources, with the additional aspect that queries often happen over

a limited window of recently seen data, instead than over the complete historical data. In comparison to the processing of batch-oriented data, streaming data exhibits varying characteristics and posses several challenges, requiring specialized efficient systems to handle them [117]. These specialized systems known as Stream Processing Engines (SPEs) are designed to process streaming data as it arrives on-the-fly without the necessity of storing it. Additionally, these SPEs must be able to fulfill the eight requirements of stream processing [47] to be able to efficiently process high-volume data streams with low latency and exhibit varied applicability. The STREAM [48] project at Stanford is a representative research initiative proposing a data stream management system (DSMS) to process continuous data streams. STREAM proposed as a general-purpose DSMS utilizes its own query language called as the continuous query language (CQL) [49] which extends SQL to be adaptable for continuous queries on data streams. Additionally, CQL, as a query language, is expressive enough to capture the dynamic characteristics of data streams through complex queries. Further, continuous queries specified in CQL are evaluated through their translation into a physical query plan to be executed by the DSDM. Several mechanisms, including the elimination of data redundancy, optimized operator scheduling are employed by STREAM to efficiently execute these query plans and improve the overall system performance. Additionally, STREAM compromises of StreaMon [118] for monitoring and adaptive query processing for data streams with varying characteristics. Additionally, STREAM also includes a graphical *query and system visualizer* to facilitate users to inspect the system interactively.

Storm is a popular project for stream processing [50], serving as a real-time distributed data stream processing system at Twitter. Storm is a scalable and resilient system capable of easily and efficiently analyzing social media streams as with Twitter and supports for real-time data-driven decisions. Supporting real-time stream analytics effectively also requires an efficient management system in place for the collection and delivery of the large amounts of streaming data generated. Apache Kafka [51] supports this requirement, with a focus on storage management and communication instead of query processing. It serves as a distributed messaging system to manage large amounts of streaming data through in-memory analytics, facilitating large data movements in a robust and scalable manner. Kafka provides support for offline as well as online analytics and also exhibits the advantage of employing a pull-based consumption model which allows an application to consume data according to its own needs without being overloaded, thus maximizing the overall throughput. Additionally, in Storm, Kafka can be utilized to pull data from message queues, to generate a stream of tuples for their further processing.

(vi) **Batch meets Streaming** Batch meets Streaming represents systems which are designed to support both batch and stream data processing. As an example, Apache Flink [39] is an open-source system providing a unified single execution model for batch and stream data processing, which can express and execute different classes of data processing applications as pipelined fault-tolerant data flows. Deployment, core runtime (the distributed dataflow engine), APIs and libraries are the four main layers that contribute towards the Flink architecture with all programs eventually compiled to a common

representation of the dataflow graph. Flink provides a sophisticated highly flexible windowing mechanism which can compute both early & approximate and delayed & accurate results in the same operation in large-scale data-stream processing. Additionally, it also supports different notions of time, providing flexibility in defining event correlations in data-stream processing. Flink also supports competitive optimized batch processing through the inclusion of specialized APIs, data structures and algorithms. Eventually, Flink as a unified platform for processing batch and stream data supports out-of-order processing, iterative processing, consistency through the maintenance of exactly-once state and provides for results with high throughput and low latency.

(vii) **Hybrids** Selection of an appropriate hybrid of systems for a large-scale data processing workflow might be beneficial in achieving high throughput. In the absence of a dynamic workflow manager, design, implementation or scaling of such a workflow requires high expertise and is often challenging and time-consuming mainly due to the tight coupling between the front-end frameworks and the back-end execution engines. In such a scenario, Musketeer [119] is the first Big Data workflow manager which decouples this coupling, providing enhanced flexibility by facilitating users to write the workflow once and dynamically map it to many different systems. Eventually, this makes the design, implementation and porting of the workflow to other systems much easier and efficient as compared to fixed single system mappings. The decoupled data processing architecture of Musketeer is divided into three layers namely, *Front-End Frameworks* which users utilize to specify their workflow using high-level abstractions, *Intermediate Representation* (IR) which dynamically converts and optimizes the specified workflow into a common form acceptable by the front and back-end, ideally in the form of a directed acyclic graph (DAG), and *Back-End Execution Engines* which based on user or automatic selection execute the jobs generated from DAG. Additionally, though Musketeer provides support for only limited front-end frameworks and back-end execution engines, the approach can be extended to support others as well, eventually facilitating mainly non-specialist users to implement flexible workflows easily.

Another workflow optimizer variant, Weld [120] provides a common runtime for cross-library optimizations in complex workflows and compromises of an IR based on parallel loops and a results merging construct called *builders* which can effectively capture loop transformations as compared to the IR utilized in Musketeer. Weld as a programming interface consists of three key components to optimize analytics applications, namely, an *Intermediate Representation* which allows libraries to express their data-parallel computation structure, a *Runtime API* which collects the IR code from different libraries for lazy execution, and a *Compiler Backend* which compiles the complete optimized Weld IR program to parallel machine code to be executed against the in-memory data of the application. Consequently, Weld enables effective composition of hybrids of libraries utilized in data analytics application enabling end-to-end optimization.

(viii) **SQL on Hadoop** SQL-on-BigData systems generally fall into two categories, namely *native Hadoop-based systems* eg., Apache Hive[4]and *database-Hadoop hybrids* eg., Hadapt[5] which integrate existing tools with the Hadoop ecosystem [3]. Additionally, analytic

appliance-based products, eg., Oracle Exadata[6]have connectors for the Big Data storage systems from where they extract data to process within their proprietary SQL engines. Figure 2.2 aptly depict the different open-source as well as commercial systems available in the SQL-on-BigData landscape. SQL for Big Data processing is of significance given its familiarity to most developers, and the ease of working with it as a high level intuitive declarative language, thus leading to an increase in the development of primarily SQL-on-Hadoop systems [58]. These systems eliminate the limitation of MapReduce framework requiring a considerable amount of programming for tasks that can be accomplished using simple SQL commands, which further adds to its complexity and limited usability for large-scale data processing [56]. Some of these systems include Drill[7], HAWQ (Hadoop with Query)[8], Hive, Impala[9], Presto[10]and Spark SQL[11]. These systems are useful for ad-hoc querying and analysis in large-scale data processing, as analysed in the study or Rodrigues et al. [52], where their main characteristics are highlighted. This study evaluates that no tool emerges as the optimal one but different tools can be utilized effectively for different requirements, eg., Hive is robust towards long-term queries and is effective for batch processing, Impala and HAWQ can be better utilized towards real-time analysis due to their fast response times and including Spark they can also be better utilized towards advanced analytics. The performance of these systems is further experimentally evaluated in follow-up studies by the authors [53, 54] using the TPC-H[12]and TPC-DS[13]benchmarks, respectively, with both evaluations proving the fact that there is no one-size that fits for all the SQL on Hadoop processing requirements. Additionally, a further experimental study [55] evaluates extensively the performance of the systems Apache Hive, Spark SQL, Apache Impala and PrestoDB using three different benchmarks namely TPC-H, TPC-DS [121] and TPCx-BB [122] considering different application scenarios and workload characteristics.

Looking into the characteristics of some of the SQL-on-Hadoop systems utilizing the MapReduce framework for query execution, Apache Hive [60] is the first work towards providing a scalable open-source data warehouse solution built on the top of Hadoop with support for queries expressed in a SQL-like query language *HiveQL*. HiveQL being flexible and extensible supports for a majority of SQL-like query constructs and also for data definition (DDL) and manipulation (DML) statements facilitating simplified data querying, creation and manipulation. A HiveQL query is basically compiled into a Directed Acyclic Graph (DAG) of MapReduce jobs for execution. The major components of the Hive architecture include, *External Interfaces* consisting of User and Application Programming Interfaces (API), the *Thrift[14]server*, the *Metastore* as the system catalog, the *Driver* managing the compilation, optimization and execution of a HiveQL statement, the *Compiler* and Hadoop as the *Execution Engine*. Yet another system Apache Pig[15]provides a high-level scripting language *Pig Latin* [59] which combines both the aspects of high-level declarative querying like SQL and low-level procedural programming like MapReduce to express large-scale data processing tasks. A Pig Latin program is usually a sequence of steps, with each step carrying out a single data transformation. Concerning the program execution, the Pig execution framework first generates a logical query plan from the Pig Latin program, which is then compiled

into a series of MapReduce jobs for execution primarily over Hadoop. Additionally, Pig also consists of its own debugging environment called *Pig Pen*, making it easier for users to build and debug their Pig Latin programs in an incremental way. On the contrary, there also exists SQL-on-Hadoop systems which do not rely on other frameworks like MapReduce for their job execution. As an example, Apache Impala[9] is a massively parallel processing (MPP) SQL query engine for data stored using Hadoop. Impala exploits a shared-nothing parallel database architecture on Hadoop through running queries using its own long-running daemons on each of the HDFS DataNode instead of relying on the MapReduce framework. There exists a main daemon process *impalad* which comprises the query planner, coordinator and the execution engine. Eventually, all queries are compiled into a pipelined execution plan. Considering experimental evaluation and benchmarking of these two variants of SQL-on-Hadoop systems, the study of Floratou et al. [58] provides performance comparisons of Apache Hive and Apache Impala using TPC-H like and TPC-DS inspired workloads. This study evaluates the significant performance advantages of Impala over Hive thus highlighting the advantages of using a shared-nothing database architecture as compared to a MapReduce based runtime for analytical SQL queries.

Apache Kudu [57] is a novel storage engine for structured data which plays a role as a middle ground between fast sequential access and fast random access. It can run in parallel with the existing HDFS installation and can be accessed via tools like Impala, Spark and MapReduce. Consequently, Kudu can be well-utilized towards fast analytics over well-structured distributed data over Hadoop. Furthermore, the different storage formats are elaborated upon in Section 2.3.3.3.

For our work, we employ Apache Spark, with a SQL-like-querying, provided by the Spark dataframe interface.

Figure 2.2: Systems compromising the SQL-on-BigData landscape [3]

(ix) **Large-Scale Graph Processing** Large-Sized graphs possess different challenges for their effective processing. Hence, efficient computation models providing competitive performance are required to process such complex large-sized graph datasets. As an example, Pregel [42] as a distributed programming framework from Google facilitates the efficient stateful processing of large-scale graphs with high scalability, performance and fault-tolerance. Pregel allows the implementation of an arbitrary graph algorithm with the input and output of a Pregel program ideally to be isomorphic directed graphs with no restrictions imposed on a specific choice of file format for these graphs. A typical Pregel computation consists of a sequence of iterations known as *supersteps* and during which a user-defined function containing the processing logic at each vertex is invoked in parallel. The communication in the entire graph occurs through its edges given the design of Pregel for sparse graphs; with the delivery of messages asynchronously in batches reducing latency. Additionally, Pregel programs are also inherently deadlock-free with their termination when all vertices are inactive with no messages in transit. Pregel offers a C++ API which is expressive and easy to program, hiding the related details of distribution. The implementation of Pregel is based on the Google cluster architecture [123] with the Pregel library dividing the input graph into partitions of a set of vertices and their edges, which are further assigned to worker machines. The master in place is responsible to co-ordinate the activities of workers. Consequently, Pregel, with its in-built features, presents itself as a simplified model for large-scale graph computing.

(x) **Deep Learning** Deep Learning models performing complex tasks exhibit large computational requirements. Training such models efficiently requires a scalable approach driven by large-scale distributed systems. As an example, Project Adam [40] is a scalable distributed deep learning training system compromising of commodity server machines, especially focusing on visual recognition tasks in a deep neural network (DNN). The high-level system architecture of Adam is based on Multi-Agent system and compromises of data serving & model training machines & global parameter server achieving both model and data parallelism. Thus, this allows for the partitioning of a large model across several machines, and hence multiple replicas of the same model can be trained in parallel using different partitions of the training data set with a common set of parameters asynchronously updated by them on a global parameter server. Adam partitions the model vertically, which minimizes the need for cross-machine communications and additionally the presence of asynchronous batched parameter updates contribute towards its multi-machine scalability. Also, multi-threaded model training is achieved on a single machine through permitting threads to update local parameter weights without locks. Additionally, Adam as a system for large DNNs training, exhibits strategies to mitigate the impact of speed variance due to slow machines and also consists of fault-tolerant strategies that can be exercised on the global parameter server.

After reviewing and providing a taxonomy for frameworks in charge of large scale processing, we consider the kinds of optimizations performed in them in the following sections. This is

informative for our research, considering that as approaches like traditional hashing provide performance optimizations for algorithms like joins; similarity-sensitive hashing can also provide for performance optimizations, which is the core of our study.

2.3.2 Optimizers for Dataflow Large-Scale Data Processing

MapReduce [44] as a framework for massive parallel processing exhibits several limitations when concerned with its utilization for efficient large-scale parallel processing and data analytics [4]. Figure 2.3 depicts these areas of improvements for MapReduce. The survey of Doulkeridis et al. [4] elaborates upon the different state-of-art research techniques focused on the optimization of these taxonomic areas.

Considering that Apache Spark[2] has emerged as a de-facto framework for Big Data Analytics [5], in this section, the optimizations that can be applied throughout the Spark Stack, enhancing its overall performance will be our primary focus.

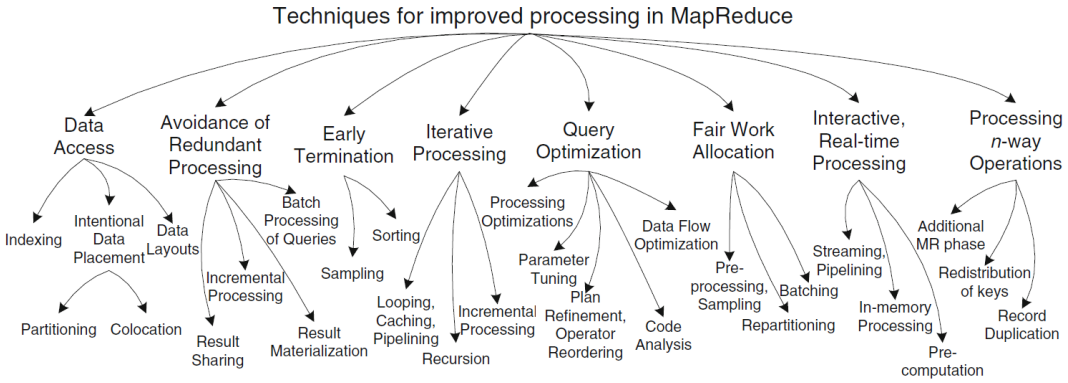


Figure 2.3: Taxonomy of Areas of Improvement for MapReduce [4]

Figure 2.4 depicts the high-level architectural overview of Spark. The main component of Spark is the *Spark core* written in Scala and offering APIs in Scala, Java, Python and R for large-scale data processing. Spark core also offers various functionalities for in-memory cluster computing including data shuffling, job recovery etc. Further, Spark core runs on different *cluster managers* namely Amazon EC2[16], Hadoop YARN [124], Apache Mesos [125] and its own standalone built-in cluster manager for job execution on various cluster resources. Additionally, Spark core can access data from any *Hadoop data source* Eg. HDFS, Cassandra[17], HBase[18], Hive[4], Alluxio[69] etc. Different *upper-level libraries* are built on top of Spark core to deal with different workloads including Spark SQL [62] dealing with structured data processing, Spark Streaming [126] for stream data processing, MLlib [7] for scalable machine learning and GraphX [127, 128] dealing with graph processing. Additionally, different *Spark packages* are external open-source packages and libraries built to work with the Spark core or the upper-level libraries, Eg. *Blink DB* as an approximate query engine over Spark SQL. Further *SparkR* [129] is an internal R package providing a R based front-end to Spark.

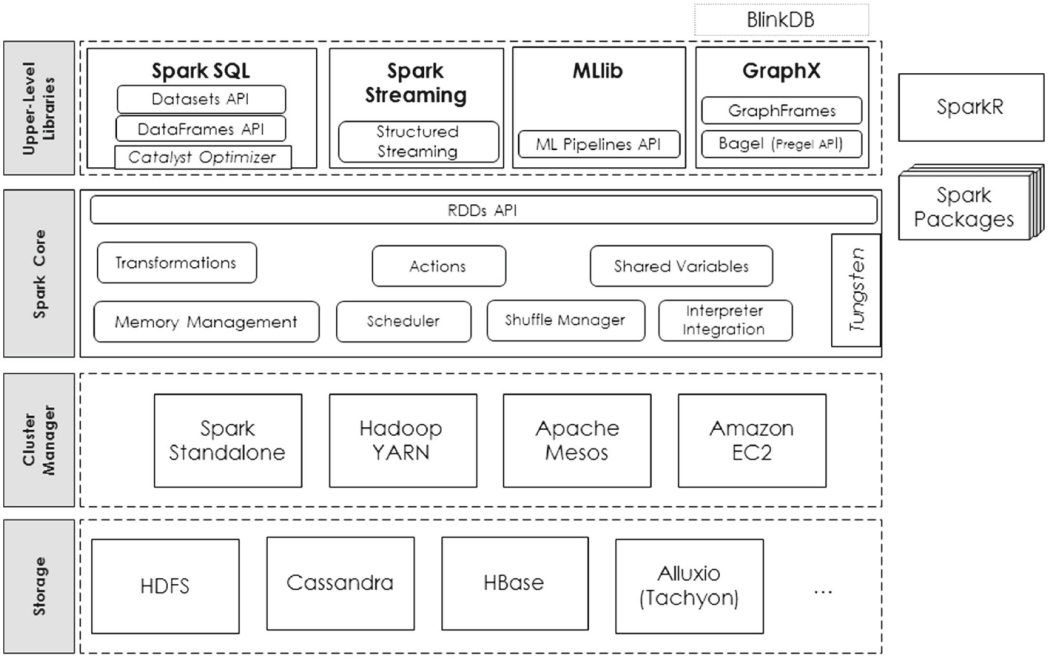


Figure 2.4: High-level Overview of Apache Spark Stack [5]

Below are some tools developed to offer optimizations for the Spark stack, increasing the overall throughput:

(i) **Project Tungsten** Project Tungsten[19]tailored towards the optimization of Spark applications focuses on improving the memory & CPU efficiency and maximum utilization of the underlying hardware [5, 61]. Tungsten as a key component of Spark's execution engine aims to achieve its goals through the incorporation of three main features namely, *Manual memory management by leveraging application semantics* which involves managing the application memory manually without the requirement of automatic memory management with Java Garbage Collector thus minimizing the memory footprint, *Data storage in CPU cache versus memory* for faster data access, *Manual code generation* which involves efficient bytecode generation as compared to standard Java VM execution, through the use of modern compilers and CPUs.

(ii) **Catalyst Optimizer** The catalyst Optimizer lies at the core of Spark SQL providing for efficient physical plan selection towards optimized job execution [62]. Catalyst is an extensible query optimizer built using the functional features of the Scala programming language supporting both rule and cost-based optimization. Catalyst at its core consists of a library which expresses the input expressions as a tree which can be manipulated using different rules. Additionally, the tree transformation framework consists of several libraries and a set of user-modifiable rules which optimize the different phases of query execution, including logical plan analysis and optimization, physical planning and code generation to Java bytecode. All of these phases are purely rule-based except the

physical planning phase, which generates and compares multiple plans executing the most cost-optimized one. Additionally, being an extensible optimizer Catalyst also offers for several public extension points to include external data sources and user-defined types allowing for advanced analytics with SparkSQL.

(iii) **RIOS (Runtime Integrated Optimizer for Spark)** RIOS [64] which can be viewed as an enhancement of Catalyst [62] is an adaptive query optimizer for Spark which selects an optimal physical query execution plan at runtime based on the idea of involving late binding of the rich and more relevant statistics gathered at runtime without the need of their upfront collection involving significant cost-overheads. RIOS specifically focuses on predicate selections for a given query to determine an optimal cost-effective join order & physical join implementation and to do so, it performs an iterative execution time strategy EGAP (Execute, Gather, Aggregate, Plan) which leverages the lazy batch strategy for building and executing query plans in Spark to derive much effective plans. The iterative EGAP strategy involves *Execution* which executes all the data and pre-partitions stages corresponding to join operations in a physical join plan, *Gather* which augments the Spark stages to collect more accurate statistics related to the join attributes in the pre-partitioning step, *Aggregate* which aggregates the collected statistics back to the scheduler (Spark Driver), *Plan* which lazily plans the next best join order and method based on the estimate Eg. Join cardinality determined given the collected statistics. Consequently, RIOS generates and modifies an execution plan at runtime progressively and greedily, thus quickly determining the optimal plan for a given query without incurring additional overheads Eg. Exploration of the entire plan space.

(iv) **Flare** Flare [65] presents itself as a new back-end for Spark which improves its relational performance to be at par with the best SQL engines and also optimizes its performance in the presence of heterogeneous workloads, based on the idea of native code generation instead of JVM code. The architecture of Flare facilitates its integration with Spark at three levels, *Flare Level 1* provides for native code compilation of the query stages within Tungsten[19] serving as a lightweight accelerator for certain queries but constrained within the Spark's runtime execution model, *Flare Level 2* provides for further optimization of the relational workloads through changes in the Spark's runtime execution model achieving significant speed-ups through the compilation of the entire Spark-generated query plans to native code eliminating Spark's runtime abstraction layers, *Flare Level 3* provides for efficient code-generation for heterogeneous workloads through the introduction of an intermediate layer between the query plans and the code generated in the form of mapping to high-performance domain-specific language (DSL) offered by the Delite [130] compiler framework.

(v) **Apache Calcite** Calcite [63] is an extensible query processing and optimization framework designed to work with heterogeneous workloads for different types of queries of which include SQL, queries over semi-structured, geospatial and streaming data. The main architectural components of Calcite consist of *Query Parser and Validator* which converts the query into a tree of relational operators, *Adapters* which provide flexibility

in determining the underlying storage mechanism, *Query Optimizer* which utilizes the relational tree and provides for cost-based and rule-based optimization, *Relational Builder Interface* which provides to express input queries with relational constructs. Calcite at its core optimizes queries in the form of relational expressions and is also able to map these back into the particular system's query processing unit. Additionally, Calcite incorporates the dynamic programming approach and prevents trapping into the local minima while minimizing the query execution costs in comparison to Catalyst [62] and hence can help to achieve better performance improvements for Spark SQL[20].

With this, we conclude our review of projects that optimize the performance of Spark applications. Generally, these projects perform query plan optimization, query compilation and handle memory management. Unfortunately, the information on the current optimizations provided by these systems is general in nature and it is difficult to understand how hashing is used internally for the optimization tasks. In our work, we consider as an optimization the algorithm restructuring brought by similarity-sensitive hashing to the tasks of supervised entity resolution. We expect that such hashing could be included, in future work, as an optimization within the developed tools.

2.3.3 Storage for Large-Scale Data Processing

In this section, we describe the different storage possibilities for optimized large-scale data processing. These are important for our research, since hashing contributes to data distribution, a core storage task. In this section, we highlight on Distributed File System (DFS) and its different types, Distributed in-memory storage and finally on the available various storage formats.

2.3.3.1 Distributed File System (HDD)

A Distributed File System (DFS) is a file system utilized towards efficient storage and retrieval of data in a distributed client-server setup. In this part, we highlight the different types of DFS along with their features. Further, we also present an example of a database designed for large-scale data processing. Specifically, here we elaborate upon Bigtable [68] which is designed for distributed storage of structured data and utilizes a specific DFS, the Google File System (GoogleFS) [67], as a building block.

Types of Distributed File System (DFS) Most of the distributed systems require coupling with distributed file systems to achieve better throughput. There are different types of DFSs [66], which cater to different requirements of the distributed systems. Figure 2.5 depicts the different types of large-scale DFS:

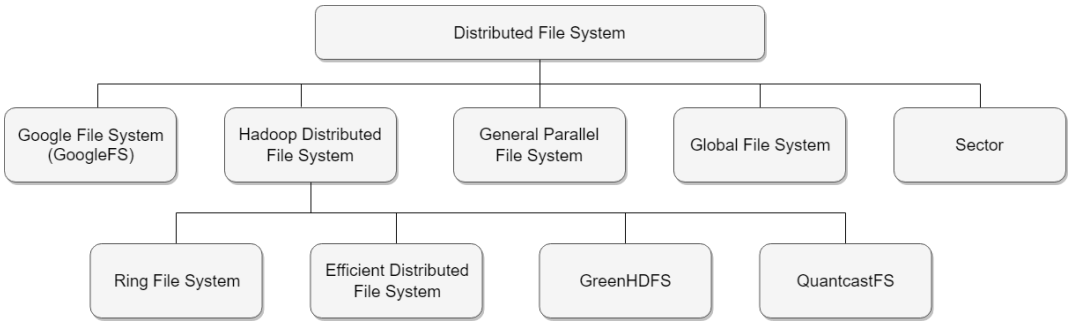


Figure 2.5: Types of Distributed File System (DFS)

(i) **Google File System (GoogleFS)** GoogleFS [67] is broadly used within Google and is optimized accordingly based on Google's data storage and usage requirements. Files in GoogleFS are divided into fixed-size 64 MB chunks accompanied by a 64-bit chunk handle, which are then distributed to the different nodes across the cluster also known as chunk servers. The master node has the responsibility of maintaining the metadata of all the chunks and their mapping to the file, and receives all the requests from the client nodes. On request, the master node sends this metadata to the client, which can then connect directly to the chunk server for data transfer. Additionally, GoogleFS can handle multiple operations occurring simultaneously on the same chunk with the help of the effectively designed locking mechanism in place. Further, to prevent overwhelming of the master node with the client requests and causing it to be a performance bottleneck, GoogleFS stores the metadata in the master's memory, which increases the overall processing speed. Data in GoogleFS is triple replicated to employ fault-tolerance and recovery in case of node failure.

(ii) **Hadoop Distributed File System (HDFS)** HDFS [131] is an open-source version of GoogleFS providing similar functionality and architecture and is based on the pattern of write-once-read-many. HDFS has different variants focusing on varied areas as described below:

- **Ring File System (RFS)** RFS [132] is considered to be a scalable and fault-tolerant DFS consisting of multiple master nodes acting as meta servers. These meta servers separately maintain part of the metadata using Distributed Hash Table (DHT) and also data is triple replicated on other meta servers. This configuration eases the load on a single master node and eliminates the single point of failure.
- **Efficient Distributed File System (EDFS)** EDFS [133] is a semi-centralized DFS and is similar to RFS in eliminating the single point of failure. In addition to the multiple master nodes, it consists of a front-end server which manages sessions and through hashing routes the requests to the master nodes. Additionally, EDFS replaces the conventional TCP protocol with another *unnamed* protocol, which is comparatively faster than TCP.
- **GreenHDFS** GreenHDFS [134] focuses on energy-conservation in a large-scale DFS. The data node in GreenHDFS is categorized into two zones, namely the hot

zone, which consists of frequently accessed or newly created files and the others being in the cold zone. Additionally, each file is associated with a temperature being of high value for the hot zone and low for the cold zone. Ideally, the rise or the fall in the file temperature depends on the access rate with a high access rate indicating a rise in temperature and the file being in the hot zone and a low access rate decreases the file temperature with the file being in the cold zone. There exists a File Migration policy which checks this file temperature and migrates the file from hot to cold zone and vice versa. These cold servers are put in a sleeping mode to achieve optimal energy savings.

- **QuantcastFS (QFS)** QFS focuses on providing better performance and cost-efficient processing in a large-scale DFS. It consists of meta servers and chunk servers but ideally one chunk server per node. Additionally, it consists of a client component which acts as an interface presenting a file system API to the other software layers and facilitates mapping of the meta server requests to the concerned chunk server and interacts directly with the chunk server.

(iii) **General Parallel File System (GPFS)** GPFS [135] is a centralized, parallel and shared-disk DFS exhibiting high scalability. Large files are divided into configurable blocks from 16 KB up to 1MB with a default of 256 KB. Sub-blocks of $\frac{1}{32}$ size of an ordinary block are used to store small files. Additionally, support for a large directory which can contain millions of files with extensible hashing to locate these files is also provided by GPFS. A distributed locking mechanism in place synchronizes the access to shared disks. Regarding the special roles played by nodes in GPFS, one node is selected as the allocation manger which maintains statistics about the free space available in all allocation regions and dynamically elected metanodes are utilized for the centralized management of file metadata. GPFS also utilizes its own fault-tolerant mechanism which involves restoring the metadata and releasing of the resources held by the failed node. It also has replacement policies in place in case it needs to replace any special roles played by the failed node.

(iv) **Global File System (GFS)** GFS [136] is a shared-disk DFS which allows direct concurrent access to the shared block storage by all nodes. The nodes in GFS are not assigned any special roles and hence, all function as peers. GFS employs a journaling file system as each node in a cluster has its own journal and the changes to the metadata of the file system are written to this journal first and then on to the file system. This helps in case of node failures to recover the consistency of the file system through replicating these metadata operations recorded in the journal. Similarly, data also can be recorded in the journal to increase the performance of this fault-tolerant mechanism.

(v) **Sector** Sector [137] is capable of being deployed over a wide area and allows handling and processing of large datasets from any location. It consists of a master node, a security server and a number of slaves nodes, with the security server being connected to the master and maintaining information on the number of slaves, file access information etc. Files are divided into Sector slices with the storage of each slice in the native file

system of the slave node as a file. To ensure data security, the slaves only function as per the master's instruction with the establishment of the connection between a client and a slave node only after the master's validation of the client request with the security server. The data in the Sector is triple replicated, thus ensuring fault-tolerance and failed nodes also can be recovered easily with the required recovery mechanisms in place. Sector utilizes UDP protocol, which is faster as compared to the conventional TCP protocol used. Additionally, Sector includes a reliable library called group messaging protocol for message passing to provide additional reliability.

Database designed for Large-Scale Data Processing Bigtable [68] is an example of a distributed storage system with the primary goal of directly supporting client applications that wish to store and manage large-scale structured data. Bigtable was mainly designed as Google's own storage solution and thereby supports a large number of Google projects and products including Google Analytics, Google Earth etc., thus providing high performance, scalability and flexibility in spite of the demanding workloads. Bigtable utilizes a simple data model consisting of a sparse, persistent and distribute multi-dimensional sorted map which is indexed using row key, column key and a timestamp, with values stored as an uninterpreted array of bytes. Row ranges in a Bigtable are referred to as *tablets* and form the basic unit of load distribution and balancing. Column keys, on the other hand, form the basic unit of access control through their grouping into sets called *column families*. Further, timestamps usually maintain the version controlling of the stored data. This simple data model provides the client with dynamic control over the data layout and format and also allows them to specify locality of the data through carefully selected data schema. Bigtable also provides a client API with different functions and features, allowing users to manipulate the stored data. Bigtable is built utilizing different building blocks from the Google infrastructure, some of them include GoogleFS to store log and data files, Google *SSTable* file format to store Bigtable data internally, Chubby [138] as a distributed lock service. Bigtable is based on the shared-nothing architecture and its implementation consists of three fundamental components namely, a library which is linked to every client to cache the tablet locations, many tablet servers responsible for handling read and write requests, tablet management and a master server responsible for functions including tablet assignment to tablet servers, tablet servers management. Additionally, Bigtable consists of various refinements in its implementation, including the use of Bloom filters [139], mechanisms to speed-up tablet recovery etc., which eventually make it a reliable structured data storage solution with high performance and availability.

2.3.3.2 Distributed In-Memory Storage

Distributed In-Memory Storage enables the stored data to reside on the main memory supporting faster retrieval and efficient utilization in response-time critical applications. In this part, we present Alluxio [69] as an example of a distributed in-memory storage system. Further, we highlight the techniques to store data facilitating its faster access by Structured Query Language (SQL) engines running on the Apache Hadoop ecosystem [70]. Additionally, we also discuss EventIndex on Apache Kudu[21]storage [71], which facilitates

efficient and faster retrieval of event data integrated with Apache Kudu's large-scale analytical features with column-oriented storage. Finally, we discuss RAMCloud [72], a data storage system hosting all of its data permanently in-memory providing high-throughput data access required for large-scale data processing.

Example System Alluxio [69] is a virtual distributed file system with memory-centric architecture enabling faster data access. Alluxio hides the storage complexity from the applications by providing itself as a data abstraction layer and enabling them to access data from different data stores seamlessly. Alluxio is based on a master-worker architecture with the workers handling the data requests and transferring the required data from the Alluxio local cached storage or the data stores to the applications requesting it, and with the master managing the workers and also the metadata requests. The Alluxio client is typically a library which can be used by the applications to interface with the servers and can be chosen from a set of different available interfaces (eg., Hadoop compatible file system interface) based on the application with minimal or no code changes. Alluxio with its features and rich set of APIs helps to decouple computation and storage effectively, and thus finds its varied applications including accelerating data analytics, simplifying data access for machine learning models.

Caching for SQL Engines on Hadoop SQL engines on Hadoop combine the convenience of querying data using SQL and the power of distributed data processing with Hadoop. However, these engines are not tightly integrated into the Hadoop ecosystem and hence caching of data for these SQL engines can significantly increase the query performance. There are primarily two methods [70] to achieve this, namely, SQL Engine Internal Caching and External Storage Systems for Caching. SQL Engine Internal Caching enables the SQL engine to implements its own internal cache providing the most control over it. This internal cache being located closest to the computation benefits from maximum performance with faster access, but faces the limitation of not being shareable typically in a multi-tenant environment. Typical frameworks that support internal caching include Apache Spark SQL[11], Apache Hive LLAP[22]. External Storage Systems for Caching as another technique utilizes an external storage system for data caching enabling data sharing and thus eliminating the limitation of the previous technique. Additionally, an independent cache provides more flexibility and manageability. This can greatly improve the performance when co-located with the computation cluster as it behaves like a shareable internal cache. Typical frameworks that support external caching include the previously discussed Alluxio [69], Apache Ignite[23]. Another alternative for external caching is the use of the centralized caching feature of the Hadoop Distributed File System (HDFS) which typically stores the user-specified files in the memory of the HDFS DataNodes. This technique is beneficial to speed up local access to the data but exhibits limited flexibility and becomes a performance bottleneck when SQL engines and HDFS are not co-located. Typical frameworks that support HDFS centralized cache include Apache Impala[9] and IBM Big SQL [140].

EventIndex on Apache Kudu Storage EventIndex is a system designed to be a complete catalogue that allows for fast and efficient searching of events from a large number of events contained in an enormous number of files and often scattered in a world-wide distributed computing system. As an example, the ATLAS EventIndex [141] designed for cataloguing all

events of the ATLAS experiment [142] and primarily using the main data store as Hadoop based on MapFiles[24]and HBase[18] allowing in-memory storage for key-value pairs providing efficient event data access. EventIndex can be effectively utilized towards event finding, retrieving, counting, and selection as well as towards data consistency checks. EventIndex exhibits a partitioned architecture which follows the data flow with components as Data Production, Data Collection, Data Storage and Query Interfaces (Command-line Interface (CLI) as well as Graphical-User Interface (GUI)). Apache Kudu implements column-oriented storage layer that complements HDFS and HBase in the Hadoop ecosystem and exhibits more efficient, flexible features towards large-scale analytics. EventIndex on Apache Kudu [71] aims towards utilizing the high-performance features of Apache Kudu which is thus able to provide a unified solution and cover more use cases as compared to the primarily used data store Hadoop based on MapFiles and HBase. Some of these use cases include data access using SQL queries, the possibility of utilization of analytical tools, faster data injection and query response. EventIndex on Apache Kudu storage thus aims to decommission the primarily used Hadoop infrastructure based on MapFiles and HBase and replace it with Apache Kudu, thus providing high-performance, scalability and flexibility.

Key-Value Store for Large-Scale Data Processing High-performance key-value storage (KVS) systems are known to be the backbone supporting many large-scale applications. RAMCloud [72] is a high-performance KVS which stores all its information completely in-memory exhibiting high-throughput and low latency for data access. A large-scale storage system using RAMCloud can be created by pooling a large number of commodity servers with the storage of data entirely in their main-memory. RAMCloud primarily relies on DRAM for storage and utilizes disk-based storage for backup and archival. RAMCloud scales automatically without additional complexity and providing applications with a unified view of the storage system. Additionally, RAMCloud also consists of different mechanisms in place to provide a high-level of durability and availability. Although RAMCloud is not as cost-effective as compared to disk-based or caching solutions, it provides guaranteed performance irrespective of the data access patterns or its locality and is well-suited towards latency-sensitive applications. Consequently, RAMCloud with its features eliminates the drawback of disk-oriented storage in-terms of latency, scalability, as well as caching solutions in-terms of durability, and presents itself to be a long-term general-purpose solution for large-scale application processing with high-throughput requirements.

2.3.3.3 Storage Formats

Storage Formats describe the layout for the data. Selecting an appropriate format is crucial for the efficient storage and maintenance of data for large-scale data processing. Data processing and management systems ideally consist of three levels, namely, *User-Interface*, which provides a medium to interact with the data, eg., SQL, *Logical Data Layout* which conceptualizes the data storage and its handling to the user eg., Data units as tables and records in RDBMS, *Physical Data Layout* which deals with how the data is physically stored and handled on a storage medium eg., Textual-based layout like CSV and JSON [6]. Figure 2.6 depicts the taxonomy connecting the logical and physical layout. It depicts tabular, tree (nested and graph) as the three fundamental logical data layouts with their further classification into

physical layouts consisting of textual and binary. These two data serialization formats are further mapped to their appropriate file formats depicting how the data is further structured and organized on a storage medium.

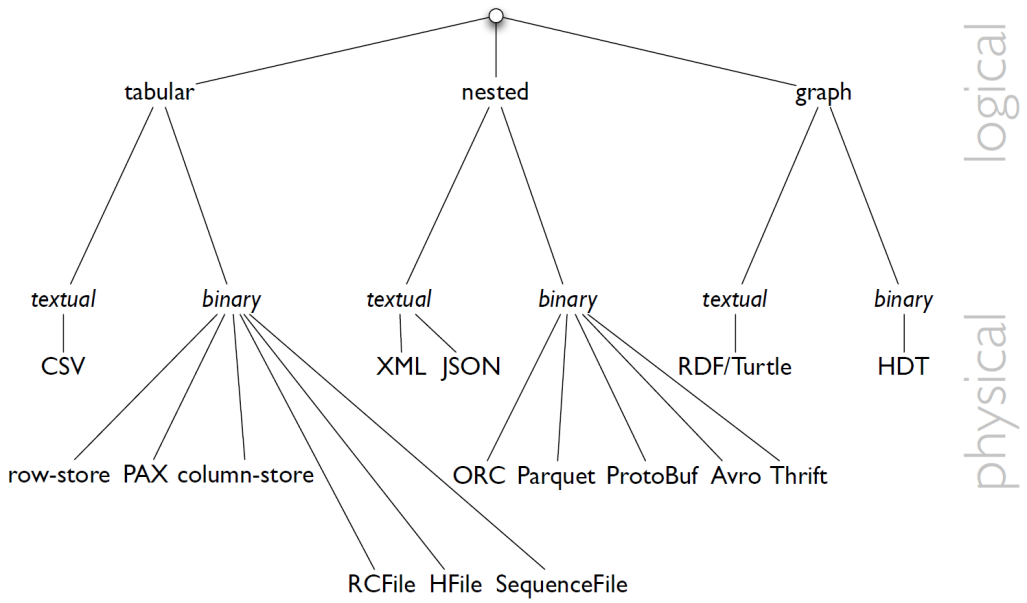


Figure 2.6: Taxonomy connecting File Format to Layout [6]

Considering the different disk-based file formats, several research works [73–75, 79, 80] highlight and describe some of the relevant ones to Big Data processing with their features as follows:

(i) **Sequential File** A sequential file is a binary file well-suited towards parallel computing, given that it can be easily split up. It utilizes the key-value format to store data. The sequential file format consists of the first three identity characters as SEQ, followed by the records and the header. The sequential file can be compressed in two ways; namely, record-wise performed on a single record and block-wise on multiple records. Further, sequential file format is supported by Apache Pig[15] and Hadoop. Additionally, Hadoop also provides for a file format called MapFile, which is a sorted sequential file maintaining indexed data and facilitating better read performance.

(ii) **Avro** Avro[25]is a row-based file format which stores the entire record one after the other. The data structure is defined using a schema usually written in JSON and contained in the metadata header of an Avro file. This schema defined can be altered as the stored data evolves, facilitating ease of stored data modification. Avro exhibits the limitation that it does not store information about the stored data values and hence cannot support data skipping.

(iii) **Record Columnar (RC)** The RC file format [143] was designed to reduce space consumption by relational data storage. The RC file format first partitions the relational data horizontally and then converts them to vertical partitions providing the column storage of a particular record together in the same data node, thus reducing tuple reconstruction costs and unnecessary column reads. RC file format, however, does not specify indexing of records and only provides for single row serializability, as a result of which various compression techniques cannot be used on it efficiently.

(iv) **Column Input (CI)** The CI file format [144] stores each column in the relational data as a separate file. This eventually helps to provide better performance through accessing only the files containing the desired columns, but it requires to have an additional implementation in place to co-locate files containing adjacent columns. This extra logic is crucial to decrease the network I/O required in record reconstruction.

(v) **Parquet** The Parquet[26]file format is well-suited towards storage of relations with a large number of columns and processing of nested data. Parquet being a columnar storage-based file format provides for a high-performance, cost-effective data storage. It stores data as one column after the other, placing data of the same type together and providing for type-based compression. Also, data is organized in row groups, which supports data skipping, scanning fewer relations in conditional query execution. Parquet organizes its data in pages containing the header, information on defined and repeated data levels and the actual data. Parquet is supported by Apache Hive[4], Hadoop, Pig, Impala, Drill[7], Spark.

(vi) **Optimized Row Columnar (ORC)** The ORC[27]file format was basically designed to eliminate the limitations of the RC file format. An ORC file is divided into a set of stripes corresponding to multiple groups of row data, with a default configurable size of about 250 MB and functioning similar to an HDFS block. Each stripe contains only an entire row data uniquely, with its column data stored as one after the another, similar to as in Parquet. Additionally, each stripe contains index information, actual data, stripe footer (containing information related to encoding and the location of the columns in the data), and with the file footer (containing information related to the data stored in the file) and a PostScript (containing necessary information to parse the file) towards the end of the file. An ORC file can be compressed in chunks facilitating efficient data skipping. ORC is more advanced as compared to Parquet given its features, including, support for transactions, better read performance as it effectively stores data statistics, efficient reduction in the data size consuming less space. ORC is further supported by Apache Hive, Pig, Hadoop.

(vii) **Albis** Albis [75] is a light-weight file format with a binary API used to store relational data on modern hardware and providing performance close to that of the hardware used. Additionally, Albis supports primitive as well as variable data types, having a simple and nested schema. Albis partitions the relational data horizontally into multiple row groups and then vertically into multiple column groups. Each column group is then considered as a separate table having its own schema where the row data is stored

one after the other in a continuous way. Albis aims to reduce the CPU costs, and hence, it stores data without any compression or encoding as a simple binary blob trading storage space to achieve better performance and generating a large-sized file. Albis also facilitates easy data management by storing the data, metadata and the schema in separate files with certain specified conventions. Unlike most of the other file formats, Albis aims to strike a balance between the CPU-I/O work division providing competitive performance on the available modern high-performance network and storage hardware.

Considering the different disk-based open-source file formats, we can observe the efficiency of columnar data storage, especially ORC and Parquet towards large-scale data processing and fast analytics [74]. In the context of open-source technologies for columnar data storage, Apache Arrow[28]provides for in-memory columnar representation, which can be used on the top of the disk-based file formats to provide fast and efficient analytical processing. Additionally, Apache Kudu in the Hadoop ecosystem provides for efficient columnar scanning and processing. Further, it provides for mutable storage in contrast to the above discussed immutable disk-based file formats requiring dataset rewriting in case of data modifications.

The determination of an optimal storage system varies for different applications as it mostly depends upon the specific use-cases catered by the application. Considering the scenario of the selection of an optimized storage format for a specific application, we elaborate upon EventIndex as discussed in Section 2.3.3.2. The comparative experimental study of Baranowski et al. [78] considers the different data formats including MapFile, Parquet, Avro, HBase and Kudu for the evaluation to optimize data storage and search performance for ATLAS EventIndex. This evaluation was carried out to determine an efficient alternative to the existing core storage implementation based on MapFiles and HBase, which exhibited concerns related to data ingestion, data duplication, data latency, random data look-up especially with the usage of MapFile format. Based on the experimental evaluation for different parameters, it was revealed that, as compared to MapFile all the other considered solutions provided faster data ingestion speed, the use of HBase or Kudu provides support for in-place data mutation making it feasible to modify data records in-place directly, use of Parquet or Kudu outputs storage efficient solutions, random data look-up time is reduced by the use of HBase or Kudu, fast and scalable data analytics is possible through the use of Parquet or Kudu. Consequently, with respect to the requirements of ATLAS EventIndex to consolidate data into a single platform, Apache Kudu was found out to be the best flexible solution. Additionally, considering a generic perspective, this study revealed columnar data stores Parquet and Kudu to be good candidates for data storage systems as they provide an effective balance between all the different evaluation parameters considered. Another experimental study, from Oles et al. [76] in the context of scalable web systems evaluated the performance of Apache Kudu in comparison to Parquet (with and without partitions), Impala-HBase and Phoenix-HBase revealing Apache Kudu to output better performance and determining itself to be a comparatively promising alternative.

Considering the performance estimation of different storage formats with the industry-accepted TPC-H benchmark[12], Pirzadeh et al. [77] evaluate some of them in the context of evaluating the performance of Big Data systems for OLAP-style workloads. The experimental

evaluation considers four Big Data systems from the different regions of the Big Data platform design space including, Apache Hive and Spark SQL (from SQL on Hadoop), System-X (a commercial parallel relational DBMS) and Apache AsterixDB [145] (from NoSQL systems). Columnar storage formats, ORC being mainly optimized for Hive and Parquet being the suggested file format for Spark were considered for evaluation. The workload consisted of variations in the schema and queries to include nested schema and comprised of a set of 22 TPC-H queries executed with different combinations of system or settings. Concerning the storage formats, this TPC-H based performance study revealed the significant performance improvements through the utilization of the optimized columnar storage formats ORC and Parquet. Additionally, it also concludes that no schema variant, storage format or system performed the best for all the queries.

To wrap things up, in this section, we have considered comprehensively the storage tools that enable large-scale processing. These span from supporting distributed disk-oriented file systems and databases, to in-memory solutions and file formats. From our review, storage file formats seem especially well-suited for fast prototyping of storage ideas, and for integrating in the future concepts for automated data skipping based on similarity hashing. We also find across all solutions a prevalent but varied use of hashing techniques to support the data distribution, and that no solution seems to be specially built to support high dimensional dense array data, like embeddings, and similarity search operations over it.

In the next sections, we shift from general frameworks to applications over them. Specifically, those related to ML and embeddings; but before that, we consider the aspect of general benchmarks for large scale processing, in order to identify the degree to which similarity search is considered as a representative application, and presenting the most closely related evaluations found in the literature.

2.3.4 Benchmarks

In the context of large-scale data processing, there is a rapid increase in the systems or frameworks catering towards efficient management & handling of Big Data, making it challenging to gauge their efficiency. In such a scenario, Benchmarks play an important role to evaluate and quantify the performance of these systems with the main aim of comparing them. The compendium study of Ivanov et al. [81] surveys existing Big Data Benchmarks. Additionally, the comprehensive survey of Baru et al. [82] categorizes open-source Big Data Benchmarks into three types namely *Micro Benchmarks*, which aim to evaluate individual system components or particular system behaviours, *End-to-End Benchmarks* which aim to evaluate the entire system with a typical application scenario for a collection of different relevant workloads, *Benchmark suites* which aim to provide comprehensive benchmarking solution by offering combinations of different micro or/and end-end benchmarks. The outcome of the two surveys [81, 82] is summarized in Table 2.3.

Further, looking into benchmarking studies that evaluate the performance of Big Data systems, Garcia et al. [88] offers an empirical evaluation and comparison of Apache Spark [45] and Apache Flink [39] for machine learning applications in batch data processing mode. This study compares the frameworks through the experimental run of three ML algorithms

namely Support Vector Machines (SVM), Linear Regression (LR) and Distributed Information Theoretic Feature Selection (DITFS) [165], on a Big dataset, evaluating the efficiency of Spark as compared to Flink with better scalability and overall faster runtimes for machine learning implementations. Additionally, Awan et al. [87] evaluate the performance of Spark from a micro-architectural perspective and report deep insights about in-memory analytics with Spark on a large-scale up server. This experimental evaluation is based on both batch and stream workloads as a subset from BigDataBench [148] & HiBench [152] for batch and as a super set from StreamBench [166] & also covers real-time analytics patterns [167] for stream processing.

As observed from our Literature survey for Big Data Benchmarks, we found that there are no clear standard benchmarks defined for the evaluation of the performance of Big Data systems in the storage and retrieval of embedding data. Hence we further elaborate upon two closely related works in this domain particularly Wang et al. [83] who discuss benchmarking array data with DSIM bench and the work of Taft et al. [84], which also considers a Big Data benchmark that looks at array data. The relevancy of these papers in the context of microarray data to understand the benchmarking for embeddings can be mapped to the definition of microarrays to be anchored arrays of short DNA elements as part of a large-scale gene expression [168], which represents similarity in its context to embeddings. In this context, Wang et al. [83] present DSIMBench as a benchmark to efficiently evaluate analytical workflows highlighting the most optimal setup in the analysis of microarray data in R. The experimental evaluation with the aim to optimize Big microarray data analysis in R using DSIMBench [83] consists of eight R workflows based on different data distributions and computational solutions measuring performance characteristics in dealing with data loading and parallel computation. Additionally, Taft et al. [84] present GenBase as a benchmark to evaluate the performance of systems for efficient Big Data data management & analytics. The benchmark evaluates on five queries including Predictive Modeling, Covariance, Biclustering, Singular Value Decomposition (SVD) [169] and Statistical Tests, with each consisting of a mix of data management, linear algebra and statistical operations workloads. Additionally, the experimental evaluation of GenBase [84] is based on a representative use case of microarray genomics data workloads but can be extended to other domains consisting of mixed workloads of data management and complex analytics.

After describing benchmarks, in the next section, we consider uses discussed in the context of embeddings.

2.4 Machine Learning and Data Management Interfaces in the Context of Embeddings

This section briefly highlights Machine Learning (ML) in the context of semantic knowledge extraction using word embeddings. It presents a view on how embeddings are utilized in practical end-to-end ML pipelines, consequently supporting different ML workloads efficiently.

Traditional relational database systems are used to process and analyse well-qualified typed entities through SQL queries which can precisely capture the syntactic relationships within

the data but leaves its semantic interpretation on the user. SQL also with its extensions is not able to provide a holistic view of the underlying relational data and thus cannot efficiently capture the semantics through the inter-or-intra-column relationships in the data, [90, 91]. Thus to provide a solution to efficiently tackle this situation, researchers have proposed the idea of a Cognitive Database [92, 170], which serves as an extension to traditional relational database systems (in fact it does not offer any implementation differences), adding *Word-Embedding* [171–174], an unsupervised neural network based approach from Natural language Processing (NLP) to extract these semantic relationships from a database table or a collection of tables. Further, Bordawekar et al. [92] detail the use of word embeddings to enable semantic queries in relational databases. Mikolov et al. put forth a method to extract the latent information from database data into a vector-representation as word embeddings [173] which captures the syntactic as well as semantic characteristics, and Bordawekar et al. [92] discuss on further using these vectors to augment traditional SQL-queries which now have cognitive capabilities also know as *Cognitive Intelligence* (CI) queries i.e., these can be used to answer advanced complex queries including semantic matching, predictive queries etc. Consequently, Bordawekar et al. [92] demonstrate how embeddings can be utilized towards enhancing database querying capabilities by providing a relational as well as a semantic view of the database. Additionally, authors [90] highlights and discusses the advantages of word-embeddings in Cognitive databases towards providing a controlled disclosure of the database information in a variety of ways. Further, Neves et al. [91] demonstrate a Spark-based cognitive database prototype with both Scala and Python interfaces and also explains with examples, different types of CI queries possible utilizing the prototype.

Concerning cognitive capabilities with Spark, Hamilton et al. [94] presents Microsoft Machine Learning for Apache Spark (MMLSpark), an ecosystem which expands the capabilities of Spark providing for a single API to execute different machine learning workloads in a variety of distributed production-grade environments. MMLSpark expands the native library of algorithms in Spark ML[43], integrates Spark with the networking language HTTP and importantly integrates Spark with Azure Cognitive Services[44]providing intelligent solutions for different ML workloads. Consequently, MMLSpark allows users to create scalable ML systems exposing them as distributed web services and precisely expands Spark's ability in the area of Deep Learning. However, this solution does not consider explicitly embedding data.

Additionally in the area of utilization of word embeddings towards enrichment of queries in Database Management Systems (DBMS), FREDDY from Gunther et al. [93] (Fast woRd EmbedDings Database sYstems), is a prototype system based on the PostgreSQL database system, which provides for simplified and wide querying based on User Defined Formats (UDFs) helping to extract semantic information from word embeddings.

Considering the aspect of evaluating the performance of text vectorization methods for the automatic measurement of semantic text similarity, Shahmirizadi et al. [15] present a comparative study with the experimental evaluation of TFIDF (Term Frequency–Inverse Document Frequency) & its related extensions, Latent Semantic Indexing (LSI) Topic model and D2V (Document to Vector) Neural model. Based on the experimental evaluation for a specific application, i.e., patent-to-patent similarity, this study evaluates that simple TFIDF is

a good choice in terms of performance and cost when the similarity differences between the vectors are relatively small. Additionally, TFIDF extensions also do not justify to be more beneficial than the simple variant in terms of performance and cost. Regarding the use of complex embedding methods like LSI and D2V, they only justify the performance and cost when the text is condensed, and the similarity detection task is relatively coarse. Further in the context of ML tasks, highlighting on the problems faced mainly by classification, skewed input datasets often consists of the unequal representation of positive and negative classes causing the class imbalance problem [175] which affects the classification accuracy. Consequently, authors like Fernandez et al. [89] highlight on the different state-of-art approaches in dealing with such imbalanced data in classification, categorized into *Data pre-processing* which involves techniques based on data under-sampling or over-sampling, *Cost-sensitive learning* which involves algorithmic modifications considering a higher significance for the positive class, *Applications on imbalanced Big Data* which involves techniques considered in addressing real-world problems in different application areas.

Embeddings in the context of Practical Machine Learning End-to-End Pipelines

Machine Learning (ML) processes are usually a series of steps that run in a sequential manner and need to be repeated several times usually to achieve the optimal model, also called as the ML pipeline. Luu et al. [95] explain practical ML with Spark using MLlib [7], aligning this to an end-to-end ML pipeline. Figure 2.7 depicts this similarity mapping of the ML pipeline consisting of the main steps to that of the MLlib pipeline.

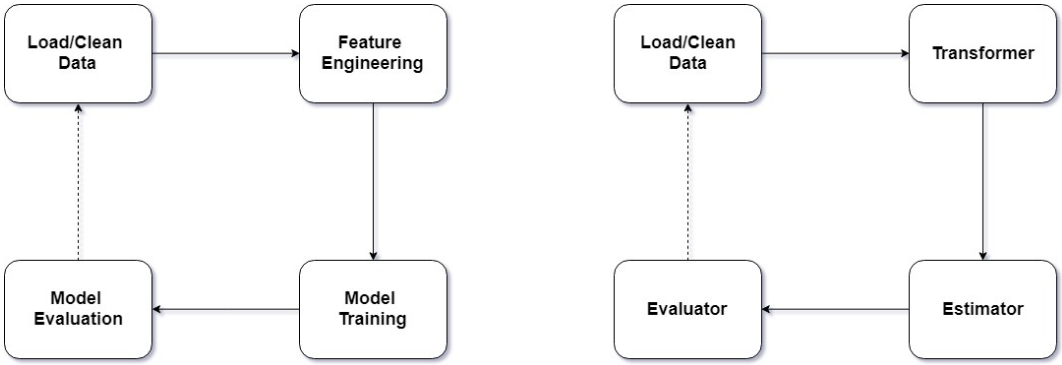


Figure 2.7: Similarity mapping between ML pipeline main steps on *left* to that of MLlib pipeline main concepts on *right* (adapted from Meng et al. [7])

The four main abstractions in the MLlib pipeline include *Transformers* which transform the input data in the dataframe to build features through data manipulation to be consumed by the ML algorithm, *Estimators* which are an abstraction for the ML algorithm which train on the data, *Evaluators* which help towards ML model tuning to achieve optimal performance and *Pipelines* which helps to easily develop and maintain the entire iterative sequential MLlib workflow. Specifically concerning our research area, in the context of Estimators, when working with text data, we have the concept of word embeddings in place. The estimator here is specifically Word2Vec, which provides for the data transformation from word tokens

to numeric vector utilizing the technique of word embeddings mapping semantically similar words to nearby points. These embeddings can be further used towards ML applications in the context of Natural Language Processing (NLP) including, word similarities, word analogies, machine translation and entity recognition. Additionally, we further highlight the different aspects of sophisticated ML using Big Data Processing systems.

Spark supports scalable and iterative machine learning through its open-source distributed ML library called MLlib [7]. MLlib includes a wide range of learning settings with several statistical, optimization, linear algebra primitives and also supports several languages including Scala, JAVA, Python, R, providing with a high-level API and thus contributing towards the development of high-performing end-to-end ML pipelines. In the area of standardizing the ML lifecycle with a goal towards simplifying and accelerating ML development, Zaharia et al. [176] present MLflow as a software platform towards streamlining of the ML lifecycle. Consequently, MLflow provides users with generic APIs that are compatible with any ML library, algorithm & programming language, thus contributing towards broad flexibility and simplifying the three main challenges in ML development towards *Experimentation, Reproducibility and Model deployment.*

Further concerning the comparative performance evaluation of Big Data Processing systems with a focus on ML, the study of Veiga et al. [98] compares and evaluates the performance of Big Data Analytics frameworks namely Hadoop, Spark and Flink suggesting significant performance improvements when Hadoop is replaced by Spark or Flink. Importantly, it considers K-means as one of the benchmarks for the performance evaluations for iterative algorithms comparing the algorithm implementation present in Apache Mahout[45]for Hadoop, with MLlib[46]for Spark and source code implementation adapted from an example in Flink. The experimental evaluation indicates the best results for K-means by Spark's optimized MLlib library, followed by Flink and then Hadoop. Consequently, Spark as a rapidly growing scalable, fast and robust system for Big Data Analytics requires a performance assessment suite to evaluate the performance of Spark-based analytics platforms. In this direction, Agrawal et al. [96] introduce a framework for the creation of a comprehensive Spark performance testing suite with a current focus on ML and graph processing workloads. Importantly for our research, work like this specifically highlights the different types of widely used ML workloads encountered in Big Data processing namely, *Logistic Regression* as an ML classifier to predict continuous or categorical data, *Support Vector Machine* which map inputs implicitly to high-dimensional feature space and conduct efficiently non-linear classifications, *Matrix Factorization* being typically used in a recommendation system as a collaborative filtering technique filling out the missing entries in a user-item association matrix, and *Random Forest Classification* which performs classification tasks using a large set of relatively simple decision trees.

After describing applications of embeddings and ML in large scale processing, in the following section, we focus on optimizations developed for array data, presenting in this context the specific approaches of similarity-preserving hashing.

2.5 Optimizations for Embeddings and Array Data

This section highlights the optimizations which can be applied for embeddings as a form of an array data. Relevantly, related work highlights on two types of optimizations namely based on *Cost Model* and *Access Path Selections*. Further, we also highlight upon high-dimensional similarity-preserving hashing on embedding data with a focus on two approaches, namely, Locality Sensitive Hashing (LSH) and Learning To Hash (L2H).

2.5.1 Optimized Management of Embeddings

In the context of optimized storage of embedding data, Svensson et al. [99] present a systematic research survey in the area of database architectures and precisely mentions SciDB [27, 29] as a new open-source scientific database idea based on the array data model to support multi-dimensional arrays with any number of dimensions. SciDB aimed towards addressing the challenge of large-scale complex scientific data analysis supports two main types of operators on array data, namely, *Structural* operating on the array structure independent of the data eg., Reshape, and *Content-Dependent* performing data-dependent content-based operations eg., Aggregation. Further, data loading in SciDB involves bulk-data loading with the storage organized as a mapping of the data to disk buckets into rectangular chunks of an array. The R-tree data structure in place keeps track of the disk locations and the bucket contents. Additionally, optimizations (discussed as open research areas in this paper) can be applied to this storage layout determining optimal bucket size, bucket compression, bucket merging etc. further leading to increased performance. Further, as another variant for array storage, TileDB [100] is a multi-dimensional array storage manager optimized for both dense and sparse arrays. Its main idea lies in the fact that array elements data is organized into ordered collections known as *Fragments* which can be dense or sparse and group contiguous array elements into fixed capacity *Data Tiles*. TileDB as a novel storage manager, consequently supports for parallelization offering faster reads and writes.

Furthermore, in the area of optimized skyline query processing [177–181] on high-dimensional data, [103] presents an efficient, scalable and robust algorithm for parallel skyline computation realized over a Hadoop MapReduce platform. This algorithm specifically addresses two important challenges, namely, *Data Skew* and *Data Stragglers* faced by skyline query processing in distributed environments over very large datasets.

Additionally, as another instance of high-dimensional vector data, Big Spatial Vector Data (BSVD) includes a wide range of spatial data types, eg., Mapping data, location-based data, etc., and can be represented by points, lines or polygon areas [182, 183]. Yao et al. [105] provide a comprehensive overview of state-of-art technologies and techniques related to data management and processing of BSVD.

Considering optimized storage layouts, hybrid layouts are increasingly popular for layout optimization to perform ad-hoc data analysis in large-scale data processing. The Hybrid layout typically divides the raw data into horizontal partitions where data is stored vertically inside each partition with support for data encoding and compression. However, these layouts require configurable parameters, eg., Horizontal partition size, which can directly

influence the query performance. Hence ATUN-HL [102] prototyped for Apache Parquet (an open-source implementation of hybrid layout in HDFS) provides for auto-tuning of these parameters to determine their best possible value based on cost-based optimization given as an input data and workload characteristics.

In the context of optimized query processing with Apache Spark on data encoded in Apache Parquet, Braams et al. [104] experimentally study the optimization technique of *Predicate Pushdown* which speeds up selective queries through pushing down filtering operations into the scan operator which is responsible to read the data and the study also presents an optimized fine-grained predicate pushdown implementation within the context of this data processing framework. Consequently, this research, with its different application scenarios demonstrates the significant increase in query performance through the exploitation of early filtering opportunities for queries involving equality and range predicates or selective joins. This technique can be reasonably applied to embedded data, with similarity-based skipping using similarity-preserving hash codes.

2.5.2 High-Dimensional Hashing

In recent years Approximate Nearest Neighbor (ANN) [107] search has become a prominent topic of research to effectively process the ever-increasing amount of data in real-world applications. Among the existing ANN techniques, hashing has become effectively popular due to its fast query speed, and low memory costs [16–24]. A hashing model takes an input data point (images, document, etc.) and outputs a sequence of bits or hashcode representing that data point. The taxonomy[47] of the hashing models based on their fundamental properties as depicted in Figure 2.8 broadly divides them into two categories, *Projection* which focus on learning a low-dimensional transformation of input data in such a way that related data points are closer together in the hashed space and *Quantisation* which focus on converting these projections into binary bits by using a thresholding mechanism. Projection can be further divided into two categories, *Data-Independent* where the hash function is randomly learned independently of the input data distribution & representative methods include Locality Sensitive Hashing (LSH) [106, 107] and its variants and *Data-Dependent* which take into account the input data distribution & representative methods include Learning To Hash (L2H) [17, 18, 108] methods. Data-Dependent models can be further categorized into *Supervised* which learn the hash function leveraging the supervised information in the form of labels to maximize the occurrence of related data points to be hashed to the same buckets and *Unsupervised* which learn the hash function without requiring the supervised label information typically through data factorizing. In comparison to data-dependent, data-independent methods can achieve comparable or even better accuracy with shorter hash codes and hence are becoming increasingly popular in real-world applications. Consequently, these high-dimensional hashing methods can reasonably be studied for their utilization towards the efficient management and retrieval of embedding data. In our thesis study, we specifically consider projection with data-independent and data-dependent supervised hashing models. Further, we elaborate more upon these hashing methods.

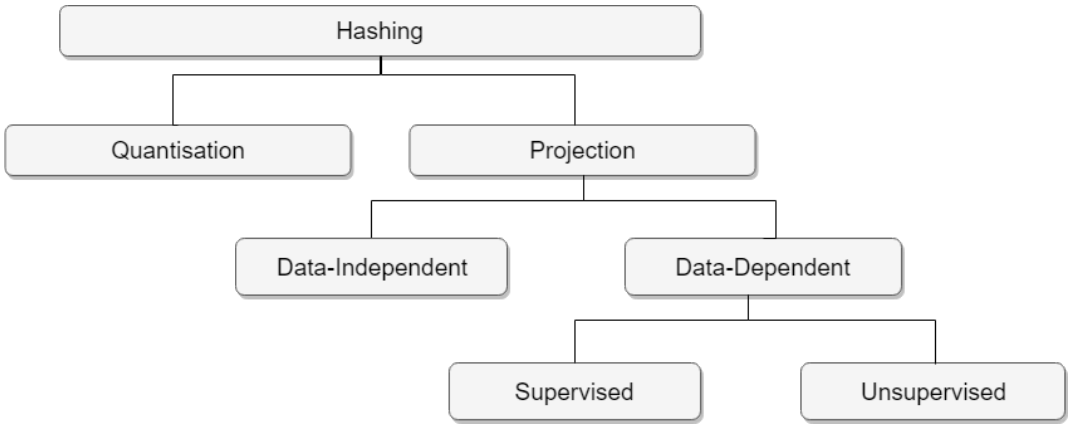


Figure 2.8: Taxonomy of Hashing Models[47]

2.5.2.1 Locality Sensitive Hashing

Locality Sensitive Hashing (LSH) [106, 107] is an approximate data-independent scalable algorithm which provides for faster comparisons between points in a very high-dimensional space through their random projection to low-dimensional bit signatures preserving their cosine distance approximately. LSH hashes input data points to buckets in such a way that similar data points are mapped to the same bucket with a high probability and data points away from each other are likely to be placed into different buckets. Consequently, LSH finds its applications in several domains some of them including, Near-duplicate detection [184], Hierarchical clustering [185], Genome-wide association study [186] etc. There exists various kinds of LSH and one among them is Binary LSH which generates binary codes for the input data points and calculates the approximate distance or similarity between them through Hamming Distance on these binary codes through mainly bitwise operations. This acts as a scalable solution in large-scale applications as these binary computations are much faster and storage-efficient. Additionally, we also have data representations in large-scale applications where the natural pairwise similarity between them is only related to the angle between them, eg., Normalized bag-of-words representation for videos, documents, images; in such cases, angular (cosine) similarity serves as similarity measurement. One of these methods is the Sign-Random-Projection LSH (SRPLSH) [187] which is a binary LSH method and provides unbiased estimates of angular similarity. However, SRPLSH suffers from the problem of large variances and thus leads to large estimation errors. Thus Super-Bit LSH (SBLSH) [8] improves upon SRPLSH to provide unbiased estimates of angular similarity with smaller variances when the angle of estimation lies between $(0,\pi/2]$ (which is maintained by real-applications as mostly they have non-negative vector representations with their orientations limited in this range). SBLSH accomplishes this by converting the random projection vectors to orthogonal vectors in batches (each consisting of L items, also called buckets) producing independent samples, as shown in Figure 2.9. The resulting bits of these independent samples are then grouped together as N-Super-Bit where N is the Super-Bit depth (also called stages). As per the experimental evaluations of Ji et al. [8], for angular similarly estimation and approximate nearest neighbor, SBLSH outperforms SRPLSH providing higher accuracy.

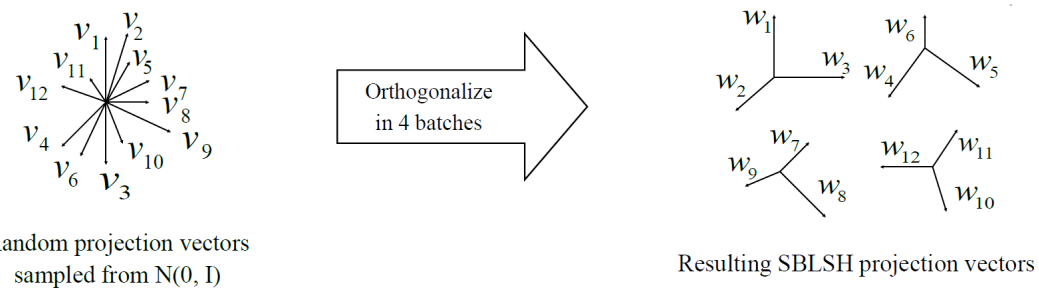


Figure 2.9: Super-Bit Orthogonal Projection Vectors of Random Projection Vectors from Normal Distribution N(0, 1) [8]

2.5.2.2 Learning To Hash

Learning To Hash (L2H)[48] [18, 108] is a set of data-dependent hashing methods which aim to learn a compact and similarity-preserving bitwise representation with shorter hash codes in such a way that similar inputs are mapped to nearby binary hash codes. L2H methods fall into two categories namely, *Unimodal* and *Multimodal* with each constituting of learning methods as *Supervised* which utilize the supervised label information to learn the hash codes during the training procedure and *Unsupervised* which only utilize the feature or attribute information of data points without any supervised label information during the training procedure. The possibility of providing the supervised label information is achieved in three different forms namely, *Point-wise labels, Pairwise labels and Ranking labels.* Some of the unimodal supervised learning methods include Semi-Supervised Hashing (SSH) [188], Minimal Loss Hashing (MLH) [189], Linear Discriminant Analysis based Hashing (LDAHash) [190], Kernel-based Supervised Hashing (KSH) [19], Latent Factor Models for Supervised Hashing (LFH) [110], FastH [22], Supervised Discrete Hashing (SDH) [23], Column Sampling based Discrete Supervised Hashing (COSDISH) [24]. Further, some of the unimodal unsupervised learning methods include Principle Component Analysis (PCAH) [191], Spectral Hashing (SH) [192], Self-Taught Hashing (STH) [193], Anchor Graph-Based Hashing (AGH) [194], Iterative Quantization (ITQ) [17], IsoHash [18], Discrete Graph Hashing (DGH) [195], Scalable Graph Hashing (SGH) [196]. Multimodal hashing constitutes of *Multi-Source Hashing* which aims to learn better binary codes than unimodal hashing through leveraging auxiliary views assuming that all the view are provided for the query and includes methods namely, Multiple Feature Hashing (MFH) [197], Composite Hashing (CH) [198], and *Cross-Modal Hashing* which returns similar items as that to the input query item and includes methods namely, Cross View Hashing (CVH) [199], Multimodal Latent Binary Embedding (MLBE) [200], Co-Regularized Hashing (CRH) [201], Inter-Media Hashing (IMH) [202], Relation-Aware Heterogeneous Hashing (RaHH) [203], Semantic Correlation Maximization (SCM) [204], Collective Matrix Factorization Hashing (CMFH) [205], Quantized Correlation Hashing (QCH) [206], Semantics-preserving hashing (SePH) [207]. *Ranking-Based Hashing* constituting L2H compromise of ranking-based methods including Hamming Distance Metric Learning (HDML) [208], Order Preserving Hashing (OPH) [209], Ranking-Based Supervised Hashing (RSH) [210], Ranking Preserving Hashing (RPH) [211] which are supported by supervised information in the form of ranking labels for the data. Further, *Deep Hashing* constituting supervised L2H utilizes

Deep Learning compromising of hashing methods including Convolutional Neural Network Hashing (CNNH) [212], Network In Network Hashing (NINH) [213], Deep Semantic Ranking Based Hashing (DSRH) [214], Deep Regularized Similarity Comparison Hashing (DRSCH) [215], Deep Hashing (DH) [216], Deep Pairwise-Supervised Hashing (DPSH) [9].

Deep Hashing

Considering deep hashing methods from the literature to be utilized in the design for our L2H technique, Li et al. [9] propose a novel deep hashing method called Deep Pairwise-Supervised Hashing (DPSH) which does simultaneously feature learning and hash-code learning using supervised pairwise labels in an end-to-end architecture utilizing feedback mechanism to learn better hash codes. This end-to-end learning framework as seen in Figure 2.10 for image retrieval has three key components, the first component is a Convolutional Neural Network (CNN) to learn image representation from pixels, the second component is a hash function which maps the learned image representation to hash codes and the third component is a loss function which measures the generated hash code in comparison with the pairwise labels. Inspired by this work, Wang et al. [10] propose a novel triplet based deep hashing method (a special case of ranking labels) which simultaneously performs image feature and hash code learning in an end-to-end manner aiming to maximize the likelihood of the input triplet labels. This method experimentally evaluates to outperform the deep hashing based on pairwise labels and pre-existing triplet label based deep hashing methods. This end-to-end learning framework, as seen in Figure 2.11 has three components, a deep neural network to learn features from images, one fully-connected layer to learn hash codes for from these features and the loss function.

Regarding the objective function to learn similarity-preserving hash codes, Zhang et al. [110] was the first to proposes a supervised hashing method, Latent Factor Hashing (LFH) to learn similarity-preserving binary codes based on latent factor models and which can be efficiently used to train large-scale supervised hashing problems. LFH models the likelihood of pairwise similarities as a function of Hamming Distance between the corresponding points. However, the optimization of this objective function is a discrete optimization problem which is not easy to solve, and hence this constraint is relaxed by transforming the discrete codes to continuous which might not achieve satisfactory performance [24]. Hence deep hashing methods utilizing LFH propose novel strategies to solve this discrete optimization problem in a discrete way, thus improving accuracy [9]. Wang et al. [10] rather solve this discrete optimization problem in a continuous way as proposed by Zhang et al. [110], but considers the quantization error induced by this relaxation in the loss function. Further, regarding an optimized loss function for pairwise supervised hashing, Zhu et al. [11] proposes a novel Deep Hashing Network (DHN) architecture which can simultaneously optimize pairwise-cross entropy loss on the learned semantic similar pairs and the pairwise quantization loss on the learned hash codes. This serves as an advantage towards better similarity-preserving learning and controlling the quality of the learned hash codes. The pipeline architecture of DHN as seen in Figure 2.12 consists of four key components, the first component is a sub-network constituting of multiple convolution-pooling layers to learn image representations, the second component is a fully-connected hashing layer to generate compact hash codes, the third component is the pairwise cross-entropy loss function, and the fourth component is

the pairwise quantization loss function. Li et al. [109] propose a novel deep hashing method Deep Supervised Discrete Hashing (DSDN) which uses both pairwise label and classification information to generate discrete hash codes in a one stream framework. The optimization process keeps this discrete nature of the hash codes to reduce the quantization error and thus an alternating minimization method is derived to optimize the loss function.

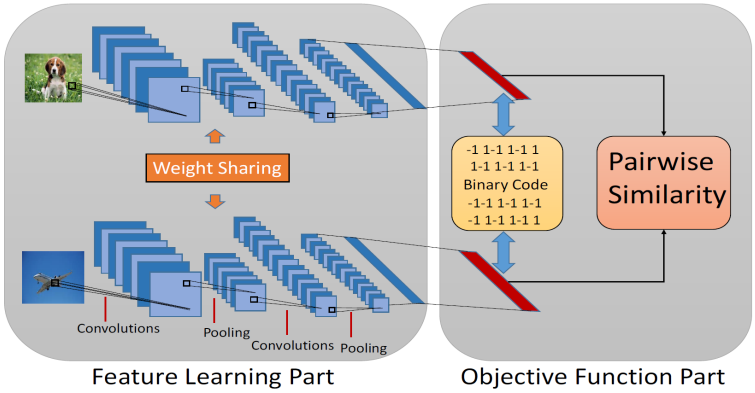


Figure 2.10: End-to-end Architecture of Deep Pairwise-Supervised Hashing (DPSH) [9]

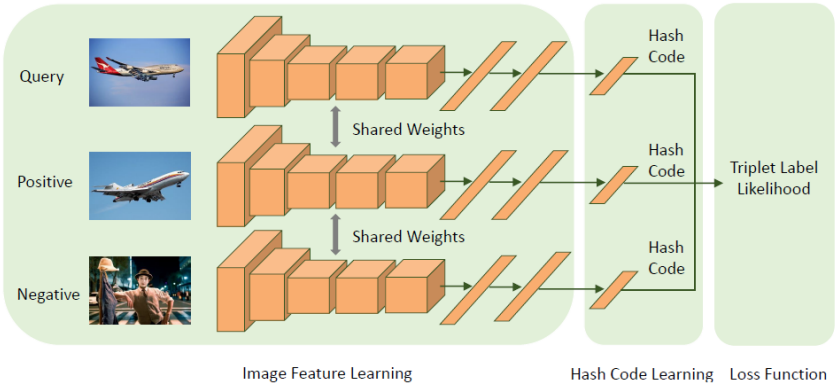


Figure 2.11: End-to-end Architecture of Triplet-Based Deep Hashing [10]

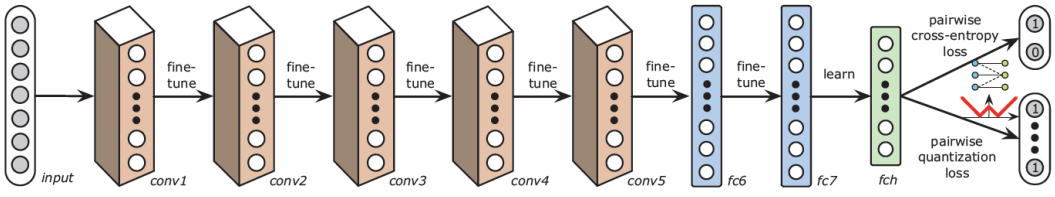


Figure 2.12: Architecture of Deep Hashing Network (DHN) [11]

In our work, we develop a solution for deep hashing, based on triplet labels.

With this section, we conclude our review of large-scale processing and management solutions for embedding data. In the following sub-section, we discuss one use case that we evaluate for embedding data: supervised entity resolution.

2.5.2.3 Supervised Entity Resolution

Entity Resolution (ER) is the process of identifying records in information systems that correspond to the same real-world entity [12]. Figure 2.13 depicts the general process flow for ER [13]. Specifically, we elaborate on these five steps mapping them to our thesis work,

- **Data Pre-Processing** We utilize and work with embedding data which highlights the semantic similarity in the data making it more manageable. Thus this step involves the processing and vectorization of the structured input datasets resulting in embedding data.
- **Blocking** Blocking splits the input data into blocks with the goal of a high-likelihood placement of the similar records in the same blocks, thus reducing the search space for the ER tasks. Consequently, given its fast query speed and low memory costs, we consider similarity-preserving hashing techniques to block these embedding data.
- **Pair-wise comparison** Pair-wise comparison evaluates the similarity between a pair of entities. Specifically, we use Cosine similarity [217–219] to compute this pair-wise similarity between the hashed vectors.
- **Classification** Once the similarity scores are calculated, we further evaluate the hashed entity pairs against the benchmark dataset i.e., Ground Truth which contains the actual correct pair mappings. This eventually trains a model (in our case, we use the eXtreme Gradient Boosting or the XGBoost[49] [220] classifier) and perform classification deciding whether the hashed pair of entities are a match or non-match.
- **Evaluation** The evaluation steps further evaluates the classification accuracy of the hashing technique. Specifically, we utilize the F1 score.

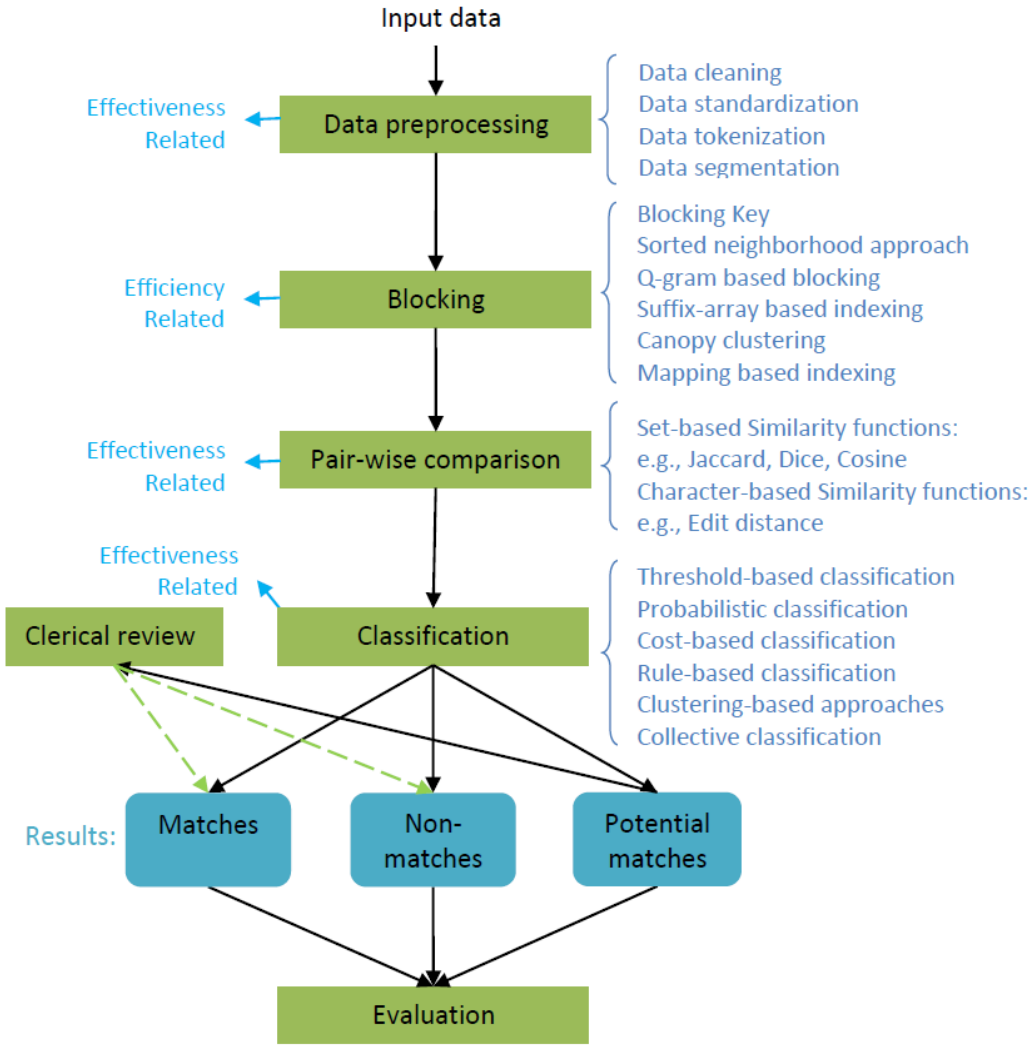


Figure 2.13: General Process of Entity Resolution [12, 13]

2.6 Summary

Summarizing, we present in this chapter a comprehensive background which provides context to the research area of this thesis. We started by briefly presenting the idea of embeddings with their applicability. In the *Large-Scale Data Processing* section, we provided a lengthy description of large-scale data processing systems, which can be considered as driving key resources in today's modern world. Additionally, we also outline the different state-of-art works to achieve optimized large-scale data processing leading to increased performance, considering optimizations and storage engines. Further, moving from the systems to the application perspective, to evaluate the efficiency of the different large-scale data processing frameworks, we outline the existing state-of-art in benchmarking and evaluation of these systems. In the *Machine Learning and Data Management Interfaces in the Context of Embeddings* section, we present the utilization of embeddings in practical

end-to-end ML pipelines towards semantic knowledge extraction and efficient support for different ML workloads. In the *Optimizations for Embeddings and Array Data* section, we focus on listing possible optimizations which can be reasonably applied for embeddings as a form of array data, among them in the *High-Dimensional Hashing* section, we explain the idea of hashing as a popular approximate nearest neighbor technique and consequently we give details on LSH and L2H. Additionally, we also elaborate upon the general process of entity resolution, as it is relevant to our evaluation. Through this hierarchical structuring of this chapter, we aim to instill in the readers a broad understanding of the research area of

this thesis slowly converging towards the main topic of research. In the next Chapter 3, we present an overview of the thesis design outlining its prototypical implementation.

[1]https://developers.google.com/machine-learning/crash-course/embeddings/
[2]https://spark.apache.org/
[3]http://www.scala-lang.org
[4]https://hive.apache.org/
[5]www.hadapt.com
[6]https://www.oracle.com/technetwork/database/exadata/overview/index.html
[7]https://drill.apache.org/
[8]http://hawq.apache.org/
[9]https://impala.apache.org/
[10]http://prestodb.github.io/
[11]https://spark.apache.org/sql/
[12]http://www.tpc.org/tpch/
[13]http://www.tpc.org/tpcds/default.asp
[14]http://incubator.apache.org/thrift
[15]https://pig.apache.org/
[16]https://aws.amazon.com/ec2/
[17]https://github.com/datastax/spark-cassandra-connector
[18]https://hbase.apache.org/
[19]https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html
[20]https://www.datascience.com/blog/grunion-data-science-tools-query-optimizer-apache-spark
[21]https://kudu.apache.org/
[22]https://cwiki.apache.org/confluence/display/Hive/LLAP
[23]https://ignite.apache.org/index.html
[24]https://hadoop.apache.org/docs/r2.7.3/api/org/apache/hadoop/io/MapFile.html
[25]http://avro.apache.org/
[26]https://parquet.apache.org/
[27]https://orc.apache.org/
[28]https://arrow.apache.org/
[29]https://amplab.cs.berkeley.edu/benchmark/
[30]http://www.tpc.org/tpcx-bb/default.asp
[31]https://github.com/intel-hadoop/Big-Data-Benchmark-for-Big-Bench
[32]http://prof.ict.ac.cn/BigDataBench/
[33]https://github.com/bigframeteam/BigFrame/wiki
[34]https://hadoop.apache.org/docs/stable1/gridmix.html
[35]https://wiki.apache.org/hadoop/Grep
[36]http://hadoop.apache.org/docs/r0.23.11/api/org/apache/hadoop/examples/pi/
[37]https://github.com/Intel-bigdata/HiBench
[38]http://sardes.inrialpes.fr/research/mrbs/index.html
[39]https://cwiki.apache.org/confluence/display/PIG/PigMix
[40]https://bitbucket.org/lm0926/sparkbench/src/master/
[41]https://github.com/SWIMProjectUCB/SWIM/wiki
[42]http://www.tpc.org/tpcx-hs/default.asp
[43]https://spark.apache.org/docs/1.2.2/ml-guide.html
[44]https://azure.microsoft.com/en-us/services/cognitive-services/
[45]http://mahout.apache.org/
[46]http://spark.apache.org/mllib/
[47]https://learning2hash.github.io/base-taxonomy/
[48]http://cs.nju.edu.cn/lwj/slides/L2H.pdf
[49]https://xgboost.readthedocs.io/en/latest/index.html

Benchmark	Description
AMP Lab Benchmark[29]	An open-source micro benchmark to measure the quantitative and qualitative analytical capabilities of data warehousing systems
TPCx-BB[30]also formerly known as BigBench[31][86, 146, 147]	An open-source end-to-end application-level benchmark to measure the characteristics of a Big Data Analytics system
BigDataBench[32][148]	An open-source benchmark suite to evaluate Big Data systems under different workloads
BigFrame[33][149]	A benchmark generator which offers benchmarking-as-a-service
CloudRank-D [150]	A benchmark suite to evaluate the performance of cloud computing systems which run Big Data applications
CloudSuite [151]	An open-source benchmark suite which analyzes and identifies performance issues in the processor's core micro-architecture and memory system organization when running Big Data cloud workloads
GridMix[34]	An open-source end-to-end benchmark to evaluate the performance of a Hadoop cluster
Hadoop Workload Examples[3536]	These ready-to-use examples in the Hadoop framework are designed to learn and micro benchmark Hadoop
HiBench[37][152]	An open-source benchmark suite for Hadoop which can evaluate the performance of MapReduce and Spark
MRBench[153]	An open-source end-to-end benchmark to measure the business-oriented performance of MapReduce systems
MapReduce Benchmark Suite (MRBS)[38][154, 155]	An open-source benchmark suite to evaluate the performance of MapReduce systems
Pavlo's Benchmark (CALDA) [156, 157]	An open-source micro benchmark which measures the performance of Hadoop in comparison with that of the traditional database systems
PigMix/PigMix2[39]	An open-source end-to-end benchmark to measure the performance of Apache Pig systems
PRIMEBALL [158]	Measures and compares the performance of Big Data parallel processing frameworks in cloud
SparkBench[40][159]	An open-source benchmark suite for Apache Spark
Statistical Workload Injector for MapReduce (SWIM)[41][160, 161]	An open-source end-to-end benchmark which provides for rigorous performace measurements of MapReduce systems by utilizing real-time workloads
TPC-H[12]	An open-source end-to-end technology-agnostic de-facto benchmark for measuring the data warehousing capability of a system
TPC-DS[13]	An open-source end-to-end benchmark which evaluates the performance of decision-support systems
TPCx-HS[42][85, 162]	An open-source first industry standard micro benchmark to stress test both Hadoop software & hardware components including the execution engine (MapReduce or Spark) and the system layers compatible with the Hadoop FS API
Yahoo! Cloud Serving Benchmark (YCSB) [163, 164]	An open-source end-to-end benchmark which measures and compares the performance of usually NoSQL database management systems

Table 2.3: Overview of Big Data Benchmarks

3. Design Overview and Prototypical Implementation

This chapter focuses on bridging the gap between the literature overview carried out and the implementation and experimentation undertaken as part of this thesis research. In this chapter, we present the design of our thesis and explain details regarding its prototypical implementation.

This chapter is structured as follows:

- Section 3.1 presents an overview of the design of this thesis.
- Section 3.2 recapitulates the research questions of focus for this thesis.
- Section 3.3 presents our proposed approach for deep hashing based on the literature as the main contribution of this thesis research.
- Section 3.4 presents an overview of the details associated with the prototypical implementation for this thesis.
- Section 3.5 presents an effective summarization of the important aspects of this chapter.

3.1 Design Overview

This section presents a high-level overview on the research area of this thesis and then maps it to the actual topic of focus for this thesis. Figure 3.1 depicts this high-level overview. In the research area of efficient management of embedding data for use in Machine Learning (ML) applications, we have different input data formats available. These data formats can be *Structured* (e.g., structured text documents, graphs) or *Unstructured* (e.g., audio, video). Using embedding methods on these data formats, such as count-based (e.g. topic models) or

prediction-based (e.g. neural models) methods (elaborated upon in Chapter 2 Section 2.2), we can obtain their vector representation in a low-dimensional embedding space, which now increases data manageability by highlighting data semantic similarity in a uniform space. As examples, in the Figure 3.1 we can see different embedding spaces for graphs[1], words[2][15] and joint/combined cases (image & audio)[3][221]. Further, in order to store and manage these embeddings efficiently, different similarity-preserving hashing techniques, such as Locality Sensitive Hashing (LSH) [106, 107] or Learning To Hash (L2H) [18, 108], can be utilized. These facilitate improved query speeds and reduced memory costs (elaborated upon in Chapter 2 Section 2.5.2), through hashing similar data together with a high probability. Such blocking of the embedded data into data groups based on their similarity can contribute to different ML tasks, even beyond just the efficiency in data management. As examples, in the Figure 3.1 we can visualize the efficiency of image retrieval tasks with nearest neighbors[4][222, 223], content-based information retrieval [5], near duplicate detection [224], location recognition [225], waveform retrieval for earthquake detection[6][226].

Mapping specifically to this research area, in our thesis, we deal with structured textual data, embedded using Facebook's pre-trained neural model fastText [25]. This was selected as a general off-the-shelf solution, which we consider to be representative of the solutions available. In order to efficiently store and manage these word embeddings, we experimentally study and evaluate two hashing techniques, namely LSH and L2H. As applications, we measure two performance factors. On one side, the performance efficiency of these hashing techniques in accelerating top-k similarity search queries on this hashed embedded data. On the second hand, the extent to which the use of the blocking performed during the hashing can contribute to improve the accuracy of a supervised entity resolution process.

The information regarding the research questions of focus for our thesis, and the prototypical implementation carried out to address them is detailed in the subsequent sections.

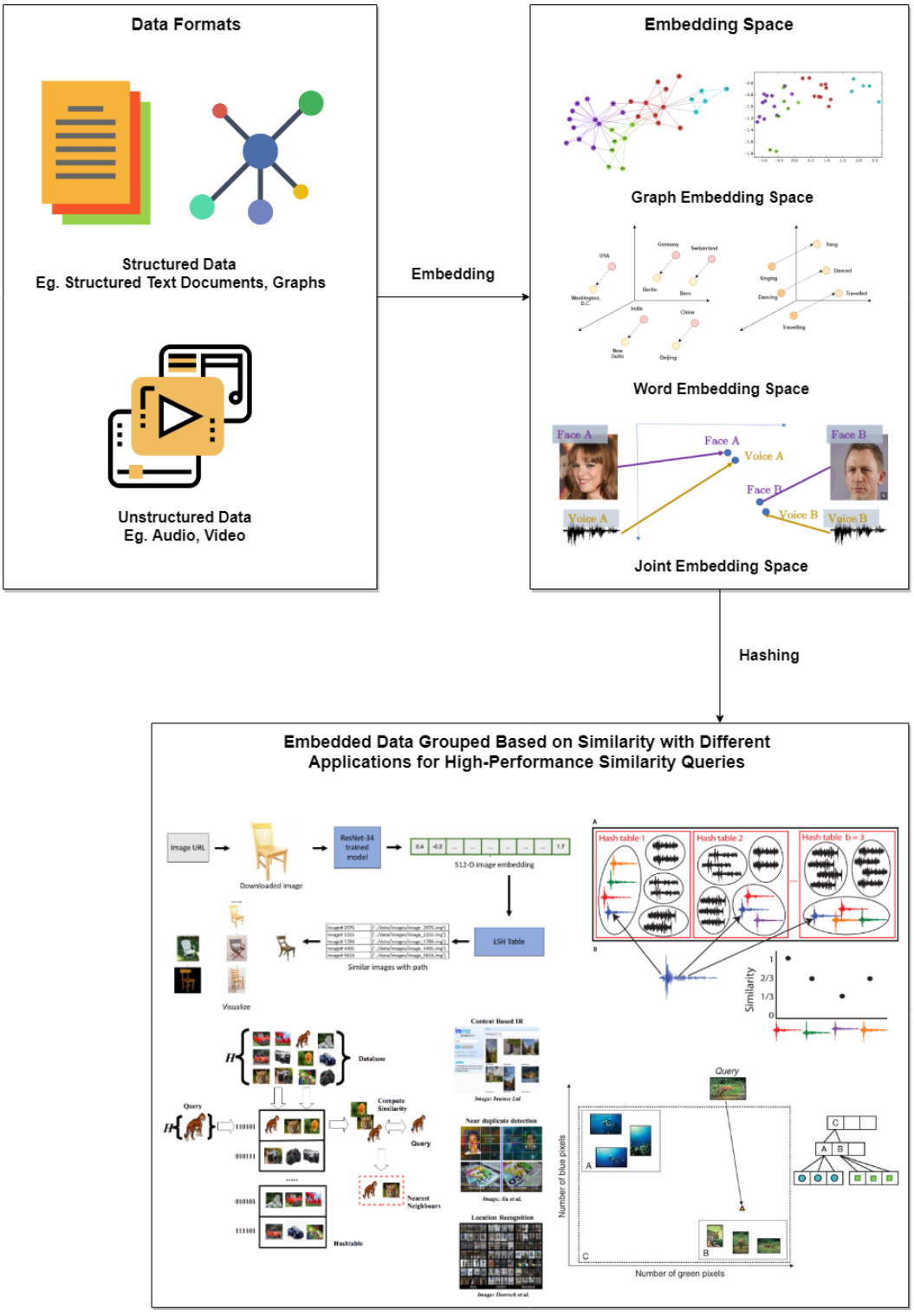


Figure 3.1: High-level overview of the research area of this thesis[16]

3.2 Final Research Questions

Based on the systematic literature review, the initial research questions were re-structured to become more concrete. Thus below are the carefully formulated final research questions which serve as the focal points to be answered through this thesis:

- **RQ**$_1$: What is the best possible coverage and block distribution, achievable using a standard high-dimensional hashing technique, Locality Sensitive Hashing? How well does the resulting data distribution contribute, as a blocking mechanism, to the performance of supervised entity resolution?
- **RQ**$_2$: How does Learning To Hash compare, with regards to blocking and coverage, to the standard high-dimensional hashing technique, Locality Sensitive Hashing? How does it compare in the task of supervised entity resolution?
- **RQ**$_3$: What is the performance (execution time) speed-up achievable by hashing with the best-observed technique, for top-k searches in a large scale processing framework? To what extent can partitioning by the corresponding hash codes affect the overall performance?

The detailed explanation regarding the evaluation metrics used in the research questions is added in Section 3.4.4.

3.3 Proposed Approach

In order to address our research questions, we can use existing state-of-the-art implementations for data-independent LSH (e.g., java-LSH[7]). Specifically we select Super-Bit (SBLSH) [8], due to its simplicity and its ability to provide unbiased estimates of angular similarity. However, at the time of this research, in the case of data-dependent supervised L2H, we found open-source implementations models tailored to image data, but no implementation easily applicable for embedding data. Therefore, a core task for our research is the development of a prototype for one general supervised hashing solution for embedding data. In this section, we discuss the design of such a solution.

We develop a novel supervised hashing solution based on neural networks. This method relies on a Deep Hash Neural Net (DHNN), as seen in Figure 3.2. This is influenced by the work of Wang et al. [10], which exemplifies research on triplet label based supervised deep learning. Triplet labels are a special form of ranking labels, for training purposes, containing richer information than pairwise labels, and which can be naturally decomposed into three pairwise labels. Further, the use of triplet labels in the learned hash code space facilitates the learning of codes that keep proximity of positive inputs and far distance for negative inputs, helping the notion of relative similarities among the inputs. Next, we describe our approach, establishing the formal problem definition and the deep hashing technique towards learning the hash function.

3.3.1 Problem Definition

Given two embedded datasets, each consisting of input word embeddings related to a given entity identified by a key (ID), and a similarity ground truth consisting of the pairs of keys (IDs) for similar entities, acting as the provider for the supervised label information. The triplet labelled examples for the supervised training using this data can be defined as, $\mathcal{T}$ = {($P1_0$, $P2_0$, N_0),....,($P1_n$, $P2_n$, N_n)} where n denotes the training batch size, and within the parenthesis we have the training examples, each formed by a random selection of two similar items (one for each dataset) as P1 & P2 and one dissimilar item as N (from either dataset), based on the labelled ground truth dataset. Based on these triplet labelled examples, the corresponding embedded data is passed as an input to the network in the form of three arrays of which the first two arrays contain the similar items and the third array contains the dissimilar items. These arrays are filled from positions 0 to n = training batch size, forming a batch of points. The items in the corresponding positions of the three arrays conform a triplet, for training purposes. Considering such information, our goal is to learn a hash code b_n for each input embedded item where b $\in$ {-1, +1}^{L} and L is the provided hash code length. The generated hash codes $\mathcal{B}$ = $\{b_n\}_{n=0}^{batchSize}$ should satisfy as much as possible all the triplet labels $\mathcal{T}$ in the hamming space making the hamming $distance_H(b_{P1n}, b_{P2n})$ as small as possible in comparison to the hamming $distance_H(b_{P1n}, b_{Nn})$ and to the hamming $distance_H(b_{P2n}, b_{Nn})$. Consequently, the aim is to learn a hash function $\mathcal{H}$ to map word embedding data to hash codes, keeping close the similar items and keeping far the dissimilar items.

3.3.2 Learning the Hash Function

For this task, we prototype a Deep Hash Neural Net (DHNN) to learn the hash function. We show an overview of our method in Figure 3.2. It consists of three key components, namely the Embedded Data Input, the Hash Code Learning, and the Loss Function.

Embedded Data Input This component consists of the pre-embedded vectors supplied as input, corresponding to the triplet labelled examples formed based on the ground truth dataset. Thus the training input is in the form of two positive (similar items), and one negative (dissimilar items) arrays filled from 0 to n = training batch size and supplied to the network (for training) one batch of examples at a time. Feature Learning improving the hashing performance can be optionally incorporated as an end-to-end deep hashing method, in order to slightly improve the embedding themselves throughout the training. Though feature learning is commonly added to the proposed end-to-end hashing method [9, 10], we do not study such improvement in our model. In our scenario, the embedding data input can be visualized as an output of the feature learning using Facebook's pre-trained neural model fastText [25] on the raw input datasets.

Hash Code Learning This component is designed to learn the hash codes corresponding to the input embeddings data. Specifically, we use a fully-connected neural architecture, with an input layer as large as the intended input, a single hidden layer with a total number of 5120 neurons, and an output layer with the size of the target hash code. We apply the Rectified Linear Unit (ReLU) activation function in the hidden layer, but in the output layer, there is no activation applied, but we clip the values to the range (-1, +1). After the training

is done, to create actual hash codes, we use the sign activation function, to map them to 0s and 1s.

Loss Function This component is the key component for the training procedure. It evaluates the likelihood of the learned hash codes to that of the triplet labelled input items. The goal of this loss function is to guide the optimization of the neural network weights, such that given an input example it leads to a hash code that minimizes the distance to the hash code of the similar input example, and increases the distance to the dissimilar example. To this end, the training proceeds batch-wise, with each batch consisting of the triplet examples. As a first step, the hash codes for each item are calculated using the neural network in its current state.

Based on the likelihood of the pairwise similarities through Latent Factor Hashing (LFH) [110] and influenced from the work of Li et al. and Wang et al. [9, 10], we define the loss function as follows,

Given two hash codes b_i and $b_j \in \{-1,+1\}^L$ from the set of hash codes $\mathcal{B}$ and L as the provided hash code length, Θ_{ij} denotes half of the inner product between them,

$$\Theta_{ij} = \frac{1}{2} b_i^T b_j \tag{3.1}$$

The likelihood of pairwise labels $\mathcal{S} = \{s_{ij}\}$ can be defined as,

$$p(s_{ij}|\mathcal{B}) = \begin{cases} \sigma(\Theta_{ij}), s_{ij} = 1 \\ 1 - \sigma(\Theta_{ij}), s_{ij} = 0 \end{cases} \tag{3.2}$$

where $\sigma(\Theta_{ij}) = \frac{1}{1+e^{-\Theta_{ij}}}$ is the sigmoid function and $s_{ij} = 1$ for similar labels and $s_{ij} = 0$ for dissimilar labels.

Further, the loss function can be defined as the negative log of this likelihood of pairwise labels,

$$\mathcal{L} = -\log p(\mathcal{S}|\mathcal{B}) = -\sum_{s_{ij} \in \mathcal{S}} \log p(s_{ij}|\mathcal{B}) = -\sum_{s_{ij} \in \mathcal{S}} (s_{ij}\Theta_{ij} - \log(1 + e^{\Theta_{ij}})) \tag{3.3}$$

The complete step by step derivation of this equation is given in Chapter 9.

Since in our design we consider triplet labels, Equation 3.3 for pairwise labels is modified to incorporate the triplet labels $\mathcal{T} = \{(P1_0, P2_0, N_0),....,(P1_n, P2_n, N_n)\}$. This is presented as follows, in its expanded form:

$$\begin{aligned} \mathcal{L} = -\sum_{s_{ij} \in \mathcal{S}} (& s_{P1_n P2_n}\Theta_{P1_n P2_n} + s_{P1_n N_n}\Theta_{P1_n N_n} + s_{P2_n N_n}\Theta_{P2_n N_n} \\ & - \log(1 + e^{\Theta_{P1_n P2_n}}) - \log(1 + e^{\Theta_{P1_n N_n}}) - \log(1 + e^{\Theta_{P2_n N_n}})) \end{aligned} \tag{3.4}$$

Since P1,P2 are the positive similar labels, we have $s_{P1_nP2_n}$ = 1 and P1,N & P2,N being the dissimilar labels, we have $s_{P1_nN_n}$ = 0 & $s_{P2_nN_n}$ = 0. Substituting theses values in Equation 3.4 we have the loss function to be used as,

$$\mathcal{L} = -\sum_{s_{ij}\in\mathcal{S}} (\Theta_{P1_nP2_n} - \log(1 + e^{\Theta_{P1_nP2_n}}) - \log(1 + e^{\Theta_{P1_nN_n}}) - \log(1 + e^{\Theta_{P2_nN_n}})) \quad (3.5)$$

Minimizing Equation 3.5 is a hard to solve discrete optimization problem, hence we solve this through relaxing the binary codes $\{b_n\}$ from discrete to continuous real vectors $\{u_n\}$ where $\{u_n\} \in \mathbb{R}^{L\times 1}$ as proposed in the work of Zhang et al. [110]. Therefore now Θ_{ij} can be re-defined as:

$$\Theta_{ij} = \frac{1}{2} u_i^T u_j \quad (3.6)$$

But this process induces a relaxation error which leads to reduced performance [24]. This is also known as the quantization error induced in the process to quantize the learned optimal real vectors $\{u_n\}$ to binary codes $\{b_n\}$. To reduce this impact on the performance, we propose to balance the equation by multiplying the term $\Theta_{P1_mP2_m}$ three times (since we consider triplets) to the loss function and hence deriving the final loss function to minimize as:

$$\boxed{\mathcal{L} = -\sum_{s_{ij}} (3 \times \Theta_{P1_mP2_m} - \log(1 + e^{\Theta_{P1_nP2_n}}) - \log(1 + e^{\Theta_{P1_nN_n}}) - \log(1 + e^{\Theta_{P2_nN_n}}))} \quad (3.7)$$

We should note that adding this multiplication, rather than a quantification error was found to be a solution that worked well empirically in our experiments. In spite of this, alternative solutions could also be considered.

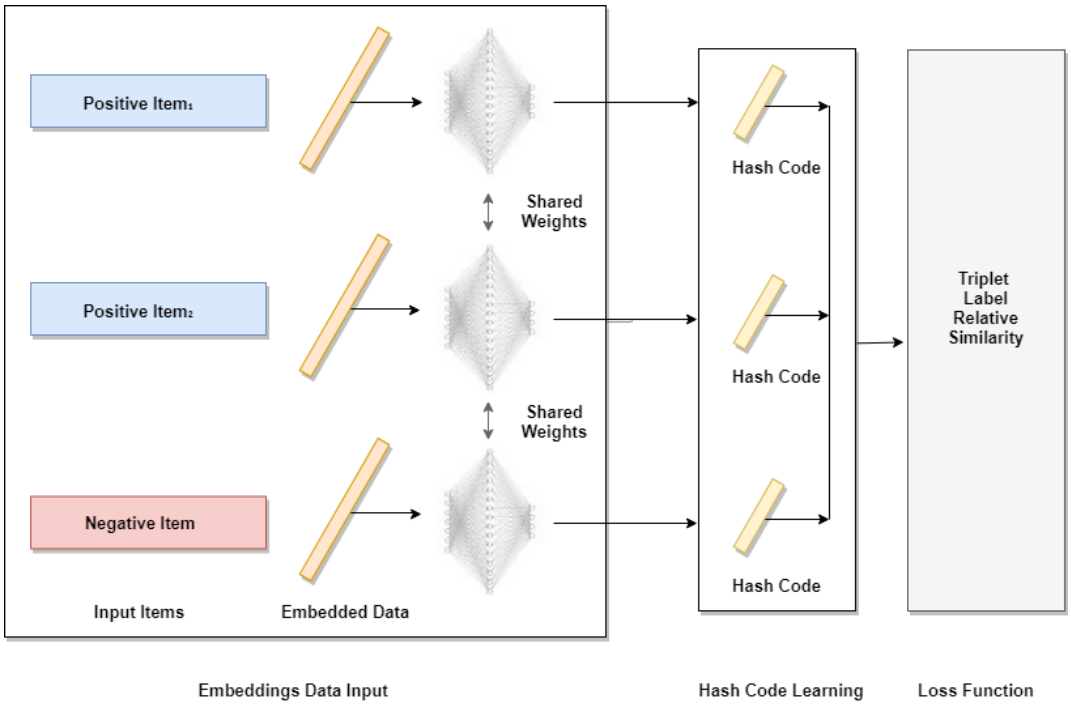


Figure 3.2: Overview of the Proposed Deep Hashing Method for Deep Hash Neural Net (DHNN)

3.3.3 Model Learning

The Deep Hash Neural Net (DHNN) as depicted in Figure 3.3 is a fully-connected deep neural network with three layers, the *Input Layer*, the *Hidden Layer* and the *Output Layer*. The size of the input layer corresponds to the size of the embeddings, i.e., 300 in case of *Single Feature Hashing* and 901 in case of *All Features Hashing*. The hidden layer is a fully-connected layer of 5120 neurons and learns the optimal hash code corresponding to the input item. Additionally, between these two layers, we use the widely used activation function *Rectified Linear Unit (ReLU)*, which exhibits the advantage of easy computation and fewer vanishing gradients resulting in better training. Finally, the output layer outputs the hash code after the loss function is minimized Equation 3.7. The size of the output layer corresponds to the specified hash code length, i.e., 12, 16, 24, 32, 48, 64, 128. Between the hidden and the output layer, we specify no activation function to maintain a linear activation. Consequently, during the network training it takes as an input array of embedded items (two positive and one negative) filled from 0 to n = training batch size one after the another and processing item by item to output the corresponding hash codes arrays (for the two positive and one negative) while minimizing the loss Equation 3.7 using the back-propagation algorithm[8]. Additionally, Table 3.1 summarizes the important network and training parameters,

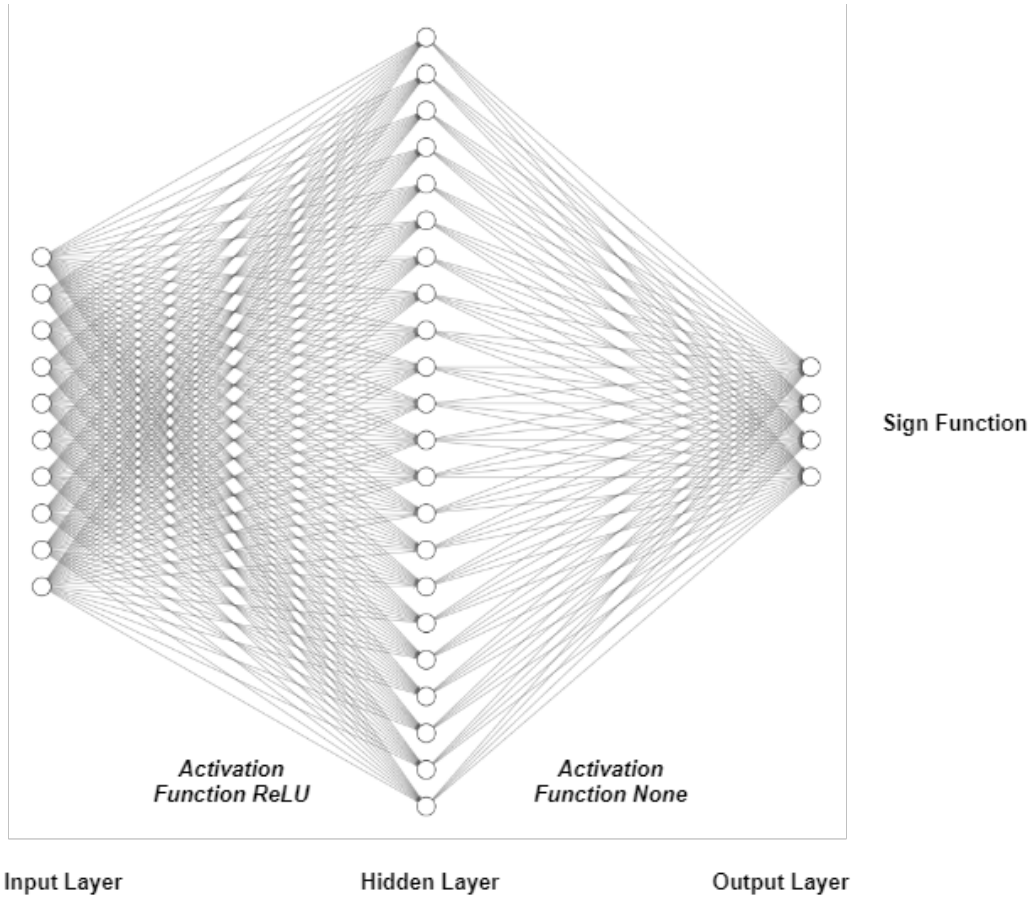


Figure 3.3: Deep Hash Neural Net (DHNN)

On each iteration, we select 50 batches and optimize for two epochs. After the completion of the training, we have the hash codes corresponding to each of the embedded input mapped to its key (ID), and then we apply the sign function on these hash codes which converts them into the format of 1s and 0s.

In our evaluation, we generate a detailed report with metrics coverage, computation (see Section 3.4.4), and coverage variations with neighboring hash matches at the end of each iteration. Hence a total of 2000 reports are created per training. The last reported values are considered for the experimental evaluation. The training converges approximately around the 1000^{th} iteration. After the last iteration, we also generate a report indicating the coverage and computation values achieved through the *Brute-Force*[9]approach acting as a baseline for the performance of the trained network to be evaluated upon. A brute-force approach typically solves a given problem by exhaustively enumerating all the possibilities, considering every possible outcome for a decision. We considered the brute-force approach to perform hashing in such a way that it generates the maximum coverage and the minimum computation value. In doing so, it exhaustively tries to assign similar items the similar hash code values thus maximizing the coverage and minimizing the computation. Additionally, we

Parameter	Values
Size of the input layer	300 or 901
Size of the hidden layer	5120
Size of the output layer	12 or 16 or 24 or 32 or 48 or 64 or 128
Activation function between the input and the hidden layer	Rectified Linear Unit (ReLU)
Activation function between the hidden and the output layer	None (Linear Activation)
Number of iterations	2000
Number of batches	50
Batch size	32
Number of epochs	2

Table 3.1: Deep Hash Neural Net (DHNN) Network and Training Parameters

also evaluate, for some cases, the trained network performance with the baseline Brute-Force approach.

We include our reference implementation in Chapter 10.

3.4 Prototypical Implementation

This section presents an overview of the essential details as part of the prototypical implementation for this thesis: the experimental setup, the input datasets, their processing, and evaluation metrics considered.

3.4.1 Experimental setup

We utilize the following configurations for our prototypical implementation:

Machine Configuration

- **Operating System** Microsoft Windows 10 Home 64-bit (Version 10.0.17763, Build 17763)
- **Processor** Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz, 1992 Mhz, 4 Core(s), 8 Logical Processor(s)

- **Memory** 16,0 GB RAM
- **Spark Cluster** 10 Spark Clients with a total 84,0 GB cluster memory

Programming Framework

- **Programming Languages** JAVA SE (Version 11.0.2), Python (Version 3.7.2), R (Version 3.5.2)
- **Programming Tools** Eclipse IDE 2018-12 (Version 4.10.0), PyDev - Python IDE for Eclipse (Version 7.2.0), RStudio (Version 1.1.463)
- **Libraries** fastText [25], java-LSH[7], Graph 3D - Playground[10], tidyverse, ggplot2, ggrepel, XGBoost[11][220], TensorFlow[12], Spark SQL[13]

3.4.2 Input Datasets

For our experimental study we primarily choose three publicly available structured dataset pairs from different domains with varying size and attributes as depicted in Table 3.2. These datasets are for product data or citations.

Dataset	Domain	Embedded Size	Features Used	Features
Amazon-Google[14][227]	Software	16,000 KB	5	ID, Name, Description, Manufacturer, Price
DBLP-ACM[14] [227]	Citation	50,000 KB	5	ID, Title, Authors, Venue, Year
Walmart-Amazon [228]	Electronics	240,000 KB	5	ID, Title, Description, Brand, Price

Table 3.2: Overview of Datasets

Additionally, each dataset pair is accompanied by a *Ground Truth* dataset which consists of the mapping of the structured tuples (ID) from each of the dataset in the pair, acting as a benchmark indicator for similarity detection tasks.

3.4.3 Pre-Processing and Vectorization of Input Datasets

For pre-processing and vectorization of the data, these dataset pairs were further embedded for String only attributes except the ID field using Facebook's pre-trained neural model fastText [25], with each embedding being of 300 dimension and with the total length of the input vector being 901. We embed each word in the attributes separately, and we include as the embedded vector the sum of these embeddings, divided by the number of words in the attribute. We do this for each string attribute in the evaluated datasets. The last entry in this vector corresponds to a normalized value (either price or year, according to the dataset). For the similarity calculation between pairs, supporting the supervised entity resolution task,

we calculate the cosine similarity between vectors of each attribute, and for the last feature in the dataset, we use the absolute difference over the normalized values. As a result, the features used for the supervised entity resolution task are the three cosine similarities (for the embedded features), and one absolute difference (for the non-embedded features).

3.4.4 Evaluation Metrics

Below are the important evaluation metrics used accordingly as part of our experimental evaluation:

1. **Coverage** User-defined metric which evaluates how many key-mappings in the Ground Truth dataset actually match to the key-mappings carried out by the selected hashing technique.

$$Coverage = \left(\frac{100 * Number\ of\ Hash\ Matches}{Number\ of\ True\ Matches} \right)$$

2. **Computation** User-defined metric which evaluates how the sum of the Cartesian product of the keys in each hashed bucket relate to the overall Cartesian product, thus it captures the block distribution by the selected hashing technique.

$$Computation = \left(\frac{Keys_{Dataset_1 HashedBucket_n} \times Keys_{Dataset_2 HashedBucket_n}}{\sum_{n=B_1}^{B_n} (Keys_{Dataset_1 HashedBucket_n} \times Keys_{Dataset_2 HashedBucket_n})} \right)$$

3. **F1 score** Metric used to measure the classification accuracy of the selected hashing technique. It is the harmonic mean of *Precision* (Fraction of relevant instances among the retrieved instances) and *Recall* (Fraction of relevant instances retrieved among the total relevant instances)[15].

$$F1\ Score = 2 \left(\frac{Precision * Recall}{Precision + Recall} \right)$$

4. **Cosine Similarity** Metric [217, 218] used to calculate the similarity between the hashed vectors through measuring the cosine of the angle between them when projected in a multi-dimensional space.

$$Cosine\ Similarity\ (VectorA, VectorB) = \left(\frac{A \cdot B}{|A||B|} \right)$$

3.5 Summary

Summarizing, in this chapter, we present the design of our thesis, which aims to map the literature review carried out to the focus of this thesis research. Further, we also present the final research questions formulated to be answered as part of this thesis. We presented our proposed Deep Hashing approach, encompassing a *Deep Hash Neural Net (DHNN)*, and we highlight on its varied aspects. We also describe the important aspects related to understanding the prototypical implementation and experimentation of this thesis research area. Specifically, we present an overview related to the experimental setup, the input datasets & their processing and the considered evaluation metrics. In the next chapters Chapter 4, Chapter 5 and Chapter 6 we present and detail out our experimental evaluation results.

[1]https://ai.googleblog.com/2019/06/innovations-in-graph-representation.html

[2]https://developers.google.com/machine-learning/crash-course/embeddings/

[3]http://www.robots.ox.ac.uk/~vgg/research/LearnablePins/

[4]https://towardsdatascience.com/finding-similar-images-using-deep-learning-and-locality-sensitive-hashing-9528afee02f5,
https://media.springernature.com/original/springer-static/image/prt%3A978-0-387-39940-9%2F9/MediaObjects/978-0-387-39940-9_9_Part_Fig1-615_HTML.jpg,
http://www.seanjmoran.com/img/binary_search.png

[5]https://imense.com/

[6]https://advances.sciencemag.org/content/advances/1/11/e1501057/F7.large.jpg

[7]https://github.com/tdebatty/java-LSH.git

[8]To be precise, in our implementation with train with RMSProp, and a learning rate of 0.0001.

[9]https://en.wikipedia.org/wiki/Brute-force_search

[10]http://almende.github.io/chap-links-library/js/graph3d/playground/

[11]https://xgboost.readthedocs.io/en/latest/index.html

[12]https://www.tensorflow.org/

[13]https://spark.apache.org/sql/

[14]https://dbs.uni-leipzig.de/de/research/projects/object_matching/fever/benchmark_datasets_for_entity_resolution

[15]https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html

[16]Documents free icon made by Freepik from www.flaticon.com, Network free icon made by Smashicons from www.flaticon.com, Multimedia free icon made by Eucalyp from www.flaticon.com

4. Locality Sensitive Hashing

In this chapter, we focus on a data-independent high-dimensional hashing technique, which reduces the dimensions of the high-dimensional input data items converting them into low-dimensional versions while preserving the relative distance between them. Specifically, we focus on Locality Sensitive Hashing (LSH) and experimentally evaluate the performance of Super-Bit, an improvement of Random Projection LSH.

This chapter is structured as follows:

- Section 4.1 recapitulates the research question of focus for this chapter.
- Section 4.2 presents an overview of our data pipeline for LSH.
- Section 4.3 discusses and presents our evaluation results for LSH.
- Section 4.4 presents an effective summarization of the important aspects of this chapter.

4.1 Research Question

Recapitulating the research questions we established for this work, in this chapter, we will be addressing the following specific research question:

- **RQ**$_1$: What is the best possible coverage and block distribution, achievable using a standard high-dimensional hashing technique, Locality Sensitive Hashing? How well does the resulting data distribution contribute, as a blocking mechanism, to the performance of supervised entity resolution?

  Relevance Measuring the Coverage determining the correct mappings and Block Distribution determining the level of block distribution serve as simple base measures to evaluate a hashing technique. They can serve as indicators for how well the data is

split according to expectations, or of the amount of data needed to be scanned for top k-similarity searches.

Description This research question deals with the experimental evaluation using two user-defined metrics namely, *Coverage*, and *Computation*. Additionally, for the task of supervised entity resolution we calculate an *F1 score* measuring this accuracy of classification with the similarity measurement using *Cosine Similarity*. These evaluation metrics are elaborated upon in Chapter 3 Section 3.4.4.

4.2 Data Pipeline

In our study on high-dimensional hashing using LSH, we primarily choose three publicly available structured dataset pairs as elaborated upon in Chapter 3 Section 3.4.2.

4.2.1 Pre-Processing and Vectorization

The pre-processing and vectorization carried out on these input datasets is as described in Chapter 3 Section 3.4.3.

4.2.2 Locality Sensitive Hashing Technique

A hash-based indexing algorithm for similarity search might be preferred in Natural Language Processing, due to its simplicity and treatment of both sparse and dense vectors with ease [229]. Consequently, with this area of focus for word embeddings, we utilize the Locality Sensitive Hashing (LSH) Technique Super-Bit [8], as discussed in Chapter 2 Figure 2.9 for our experimental evaluation. Figure 4.1 shows the high-level idea of LSH with Super-Bit hashing, which first groups data points into batches (i.e., the N stages of the Super-Bit algorithm, also called Super-Bit value or depth) and then into buckets (i.e., the L projections of the Super-Bit algorithm assigned to each batch). The given example illustrates the process for 4 stages and 6 buckets. This data division attempts to minimize hash collisions in such a way that data points near to each other are placed in the same bucket together with a high probability. In the Super-Bit algorithm that we used for our experiment, both buckets and stages are related to the code length in the following way: $code\_length = stages * buckets/2$.

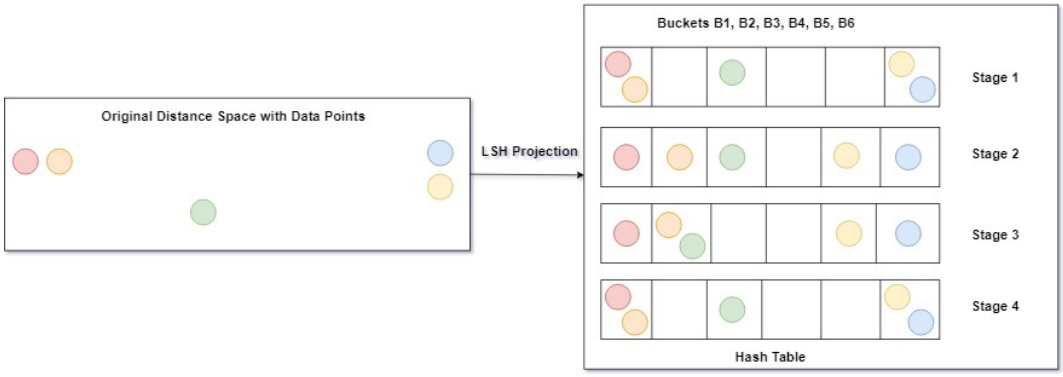


Figure 4.1: Locality Sensitive Hashing (LSH) Schematic Representation

4.2.3 Evaluation

The tuning parameters for LSH include, *Stages* denoting the number of computing iterations of the LSH hashing and *Buckets* denoting the hash buckets. We test the performance of this hashing technique utilizing *Single Feature Hashing* and *All Features Hashing* with 50 * 50 iterations for different Stages and Buckets combinations on the datasets. The Single Feature hashing attributes selected for the dataset include *Product Name* for Amazon-Google, *Paper Name* for DBLP-ACM, and *Title* for Walmart-Amazon. For each of these two variations, the optimal number of Stages and Buckets combination which output better hashing performance is discovered based on two user-defined metrics *Coverage* and *Computation* and further evaluated for classification accuracy based on the metric *F1 score* (as discussed in Chapter 3 Section 3.4.4). Consequently, we focus on the trade-off between these two values, and hence, we measure this through a trade-off analysis plot known as the *Pareto Front*. Furthermore, to test for classification accuracy for similarity search we utilize the popular *Cosine Similarity* measure [217–219] between all the related features of the hashed vectors on a selected of maximum four different Stages and Buckets combination points from this Pareto Front plot. We then utilize eXtreme Gradient Boosting or XGBoost[1][220], with default parameters, over a random train/test split of 66/33%, to test for the classification accuracy evaluated with the F1 score and eventually report these results.

Consequently, the evaluation criteria are as summarized below:

- **Benchmark** A benchmark indicator of similarity as the input Ground Truth dataset, consisting of ID-ID pairs.
- **Similarity Calculation** Similarity calculation to calculate similarity scores between the hashed vectors with Cosine Similarity.
- **Evaluation Metrics** Evaluation Metrics to assess the performance of LSH with Coverage, Computation, F1 score.
- **Tuning Parameters** Tuning Parameters to achieve optimal performance for the LSH technique with Stages, Buckets.

4.3 Results

In this section, we present the results of our experimental evaluation, answering to our formulated RQ_1. The results are categorized into two variations as *Single Feature* and *All Features* with Single Feature depicting the hashing results with Single Feature Hashing (using the first feature of each dataset) & its classification results.

There are correspondingly three plots depicted for each dataset pair, Coverage plot, Computation plot and Pareto Front plot. On analyzing these plots, we found out some generic characteristics depicted by these plots for all the datasets:

- To achieve a good hashing performance Coverage needs to be maximized and Computation needs to be minimized.

- Coverage and Computation both are high if the keys are mapped to the same bucket by the hashing, as compared with the ground truth dataset. By default, they also have a higher value when we decrease the number of buckets.
- Coverage plots have a wide spread as compared to the steep Computation plots.
- Coverage increases as we increase the number of features used for hashing in the dataset and outputs a bigger spread of possible good choices for the stages & buckets combination.
- The Coverage and Computation plots for Single Feature hashing are more stable as compared to those with All Features hashing which depict more variations.
- To facilitate evaluation on these two metrics, the Pareto Front plot depicts a trade-off between these two values with the Pareto efficient allocations for the optimum stages & buckets combination lying on the Pareto frontier.

Further, evaluating on some of these Pareto frontier solutions for their classification accuracy based on F1 score (Refer Table 4.1), we found the best performing datasets for the hashing technique ranked in increasing order to be Walmart-Amazon, Amazon-Google, DBLP-ACM as depicted in Figure 4.26.

- **Amazon-Google Dataset** The evaluation results on Amazon-Google Dataset are as below:
  - **Single Feature**
    Figure 4.2 and Figure 4.3 depict the Coverage and Computation for Single Feature hashing on the Amazon-Google dataset. Figure 4.2 shows that it was not possible to achieve a coverage of 100%, and that high coverage was only possible with very small code lengths, of size less than 50. In terms of computation, we also find similar results, as shown in Figure 4.3, with high computation overall.

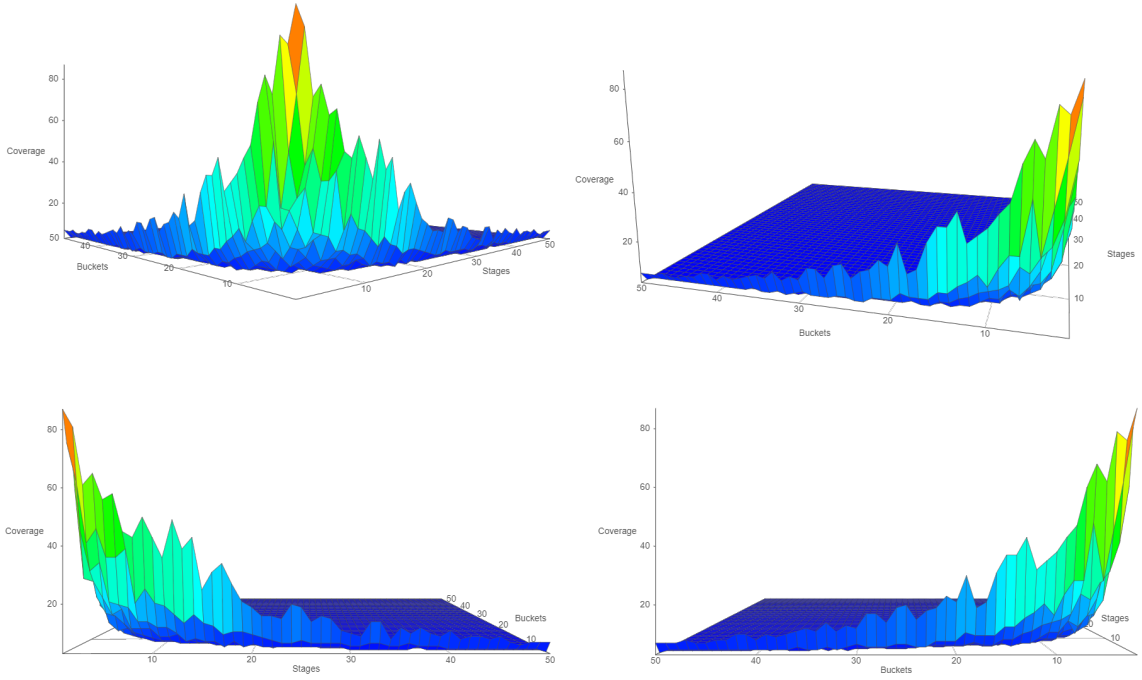


Figure 4.2: Coverage of Amazon-Google Dataset (Single Feature Hashing)

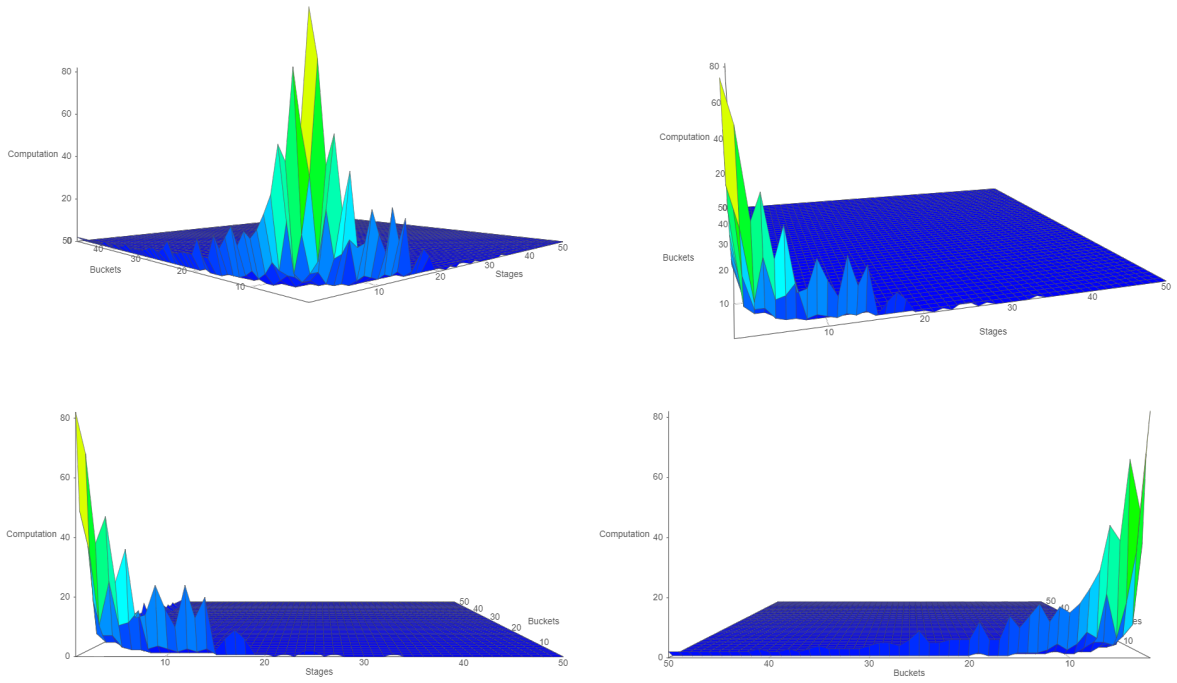


Figure 4.3: Computation of Amazon-Google Dataset (Single Feature Hashing)

For our experimental evaluation, we consider the trade-off between these Coverage and Computation values and thus Figure 4.4 provides for this trade-off analysis in a Pareto Front plot. To analyse further, we selectively label the points of interest with Computation and Coverage > = 50, as shown in Figure 4.5. Here we find the top 3 solutions, with very small code lengths of 1 or 2.

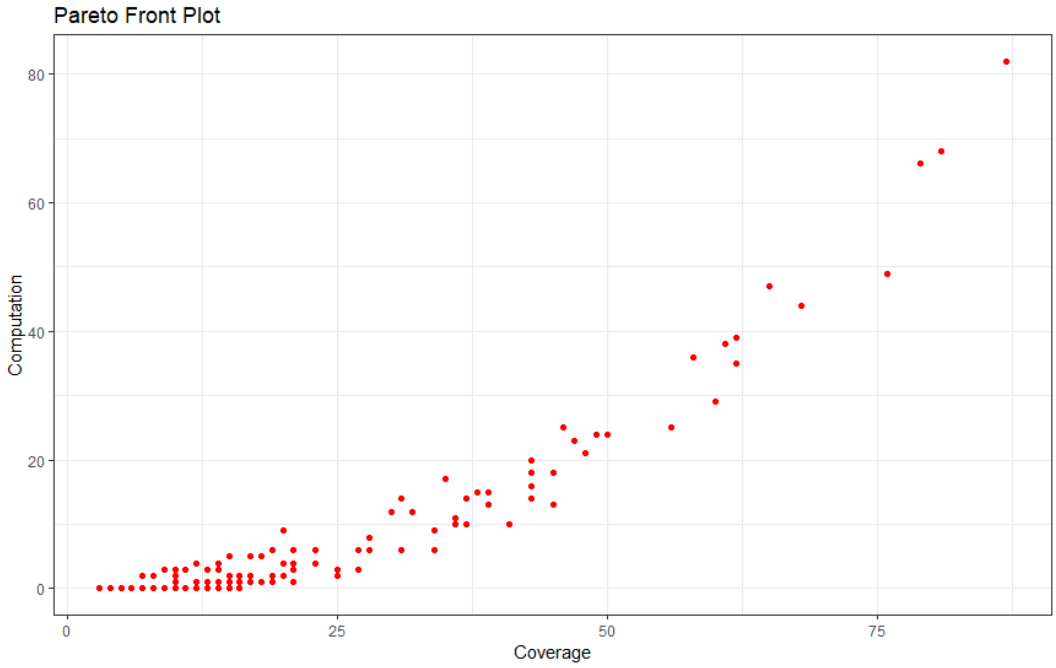


Figure 4.4: Pareto Front Plot of Amazon-Google Dataset (Single Feature Hashing)

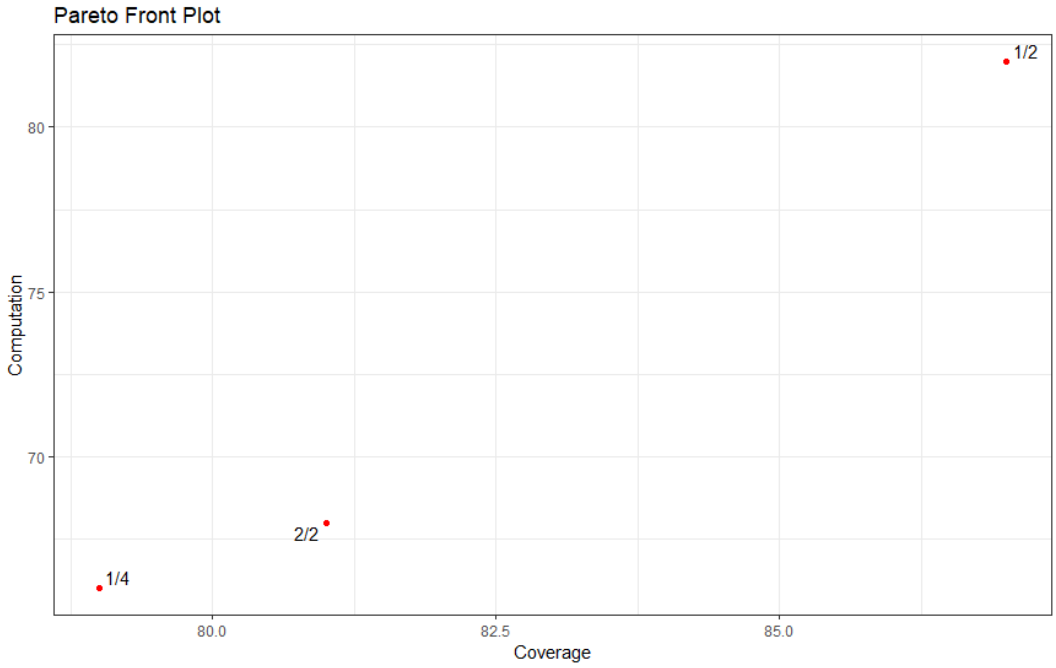


Figure 4.5: Pareto Front Plot of Amazon-Google Dataset (Selective Labelling Single Feature Hashing)

– **All Features**

Figure 4.6 and Figure 4.7 depict the Coverage and Computation for All Features hashing on Amazon-Google dataset. In contrast to the Single Feature hashing results, we observe a great variability in the space of good solutions, with the best solutions visibly located at a low value for either buckets or stages, and a higher value for the other. We observe too that there was a solution able to find 100% coverage. Regarding computation, the plots are almost indistinguishable from those of coverage, however, we can observe that computation is slightly higher on some solutions than for the case of Single Feature hashing.

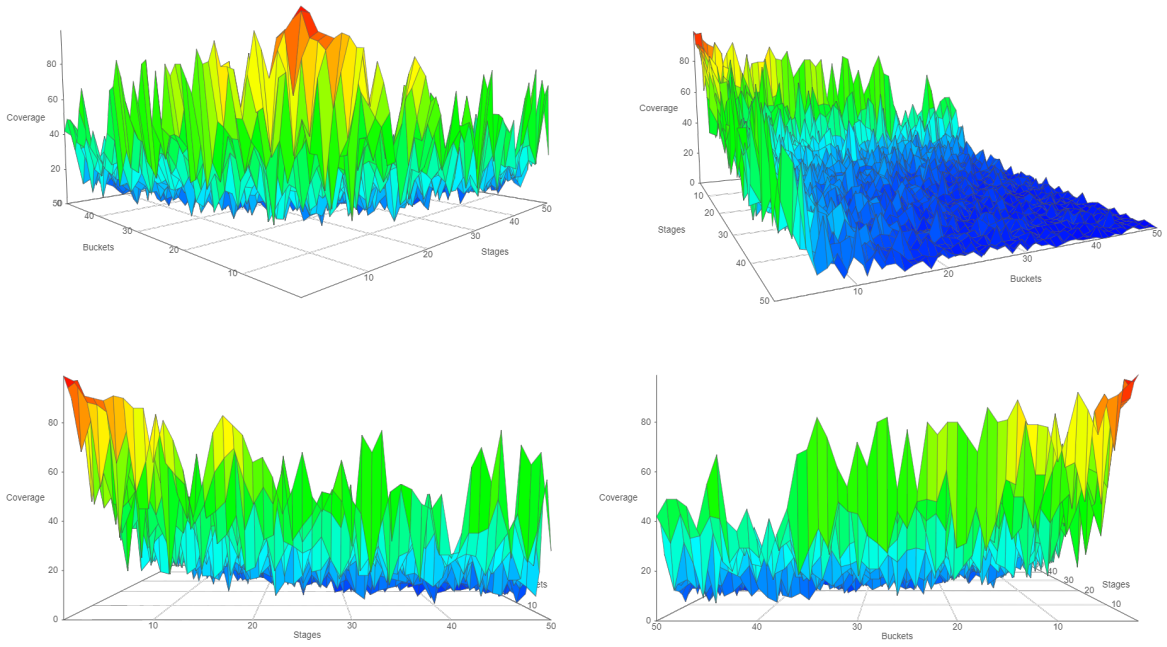


Figure 4.6: Coverage of Amazon-Google Dataset (All Features Hashing)

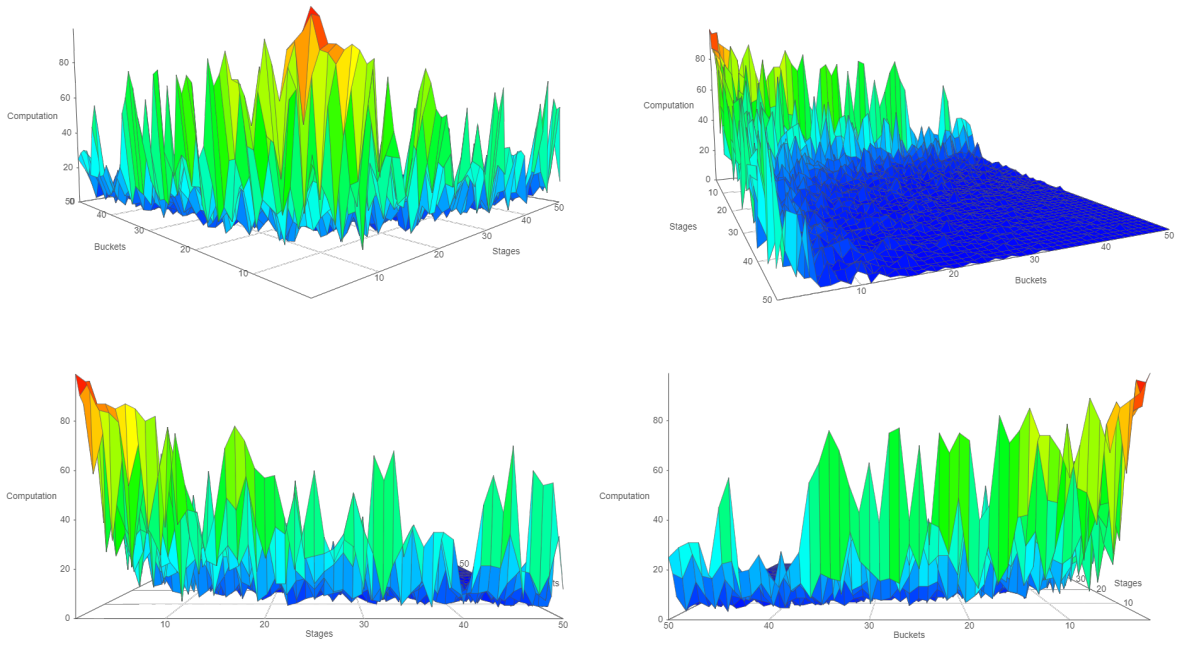


Figure 4.7: Computation of Amazon-Google Dataset (All Features Hashing)

For our experimental evaluation, we consider the trade-off between these Coverage and Computation values and thus Figure 4.8 provides for this trade-off analysis in a Pareto Front plot. To analyse further, we selectively label the points of interest with Computation and Coverage > = 50, as shown in Figure 4.9. These plots show that the solutions with the highest coverage are still with code lengths of 1 or 3 (and unfortunately with very high computation). However, plots also show a large amount of trade-off solutions that reduce the computation (improving data distribution) while keeping a relatively high coverage, while using larger code lengths (e.g., 49).

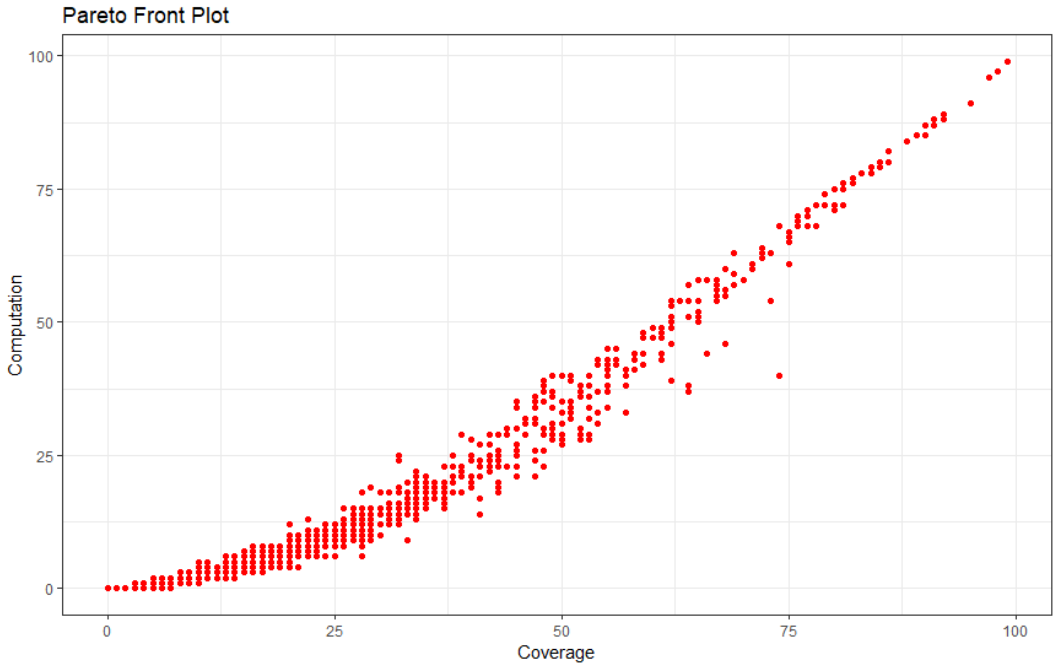


Figure 4.8: Pareto Front Plot of Amazon-Google Dataset (All Features Hashing)

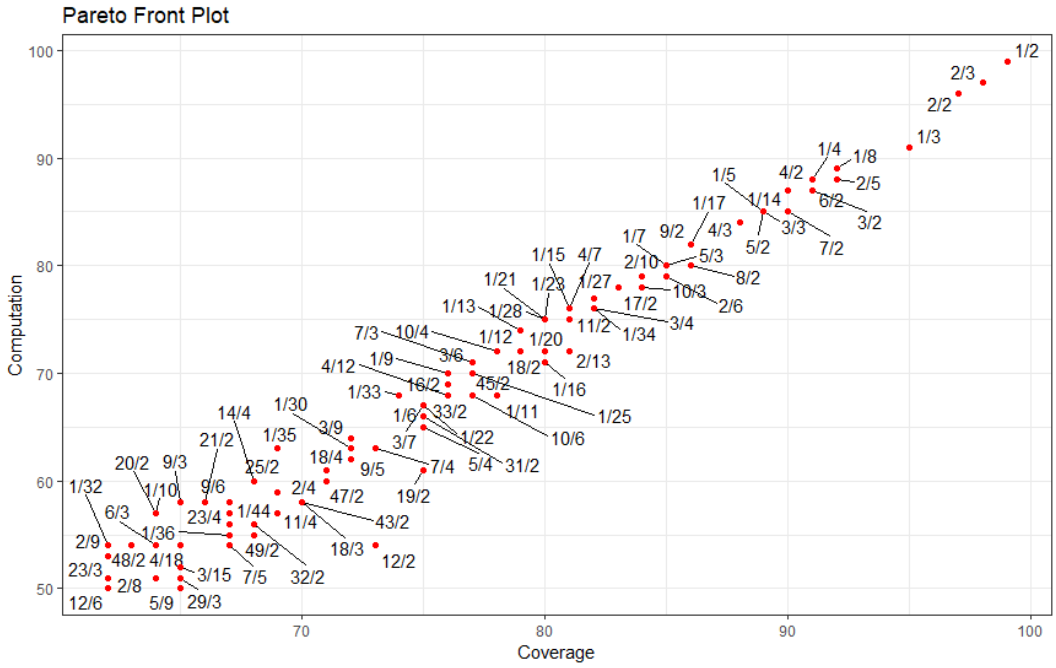


Figure 4.9: Pareto Front Plot of Amazon-Google Dataset (Selective Labelling All Features Hashing)

- **DBLP-ACM Dataset** The evaluation results on DBLP-ACM Dataset are as below:

– **Single Feature**

Figure 4.10 and Figure 4.11 depict the Coverage and Computation for Single Feature hashing on DBLP-ACM dataset. Results show that high coverage computation are only achievable with low bucket and stage values, which unfortunately lead to high computation. We observe that 100% coverage was not achievable. When compared to the Amazon-Google dataset, it is possible to note that there is a slight improvement in the coverage, with more solutions being considered as producing moderate coverage, in the space of 20/20 buckets/stages. In terms of computation, it is possible to note that the values are moderate and only peak in the 10/10 space.

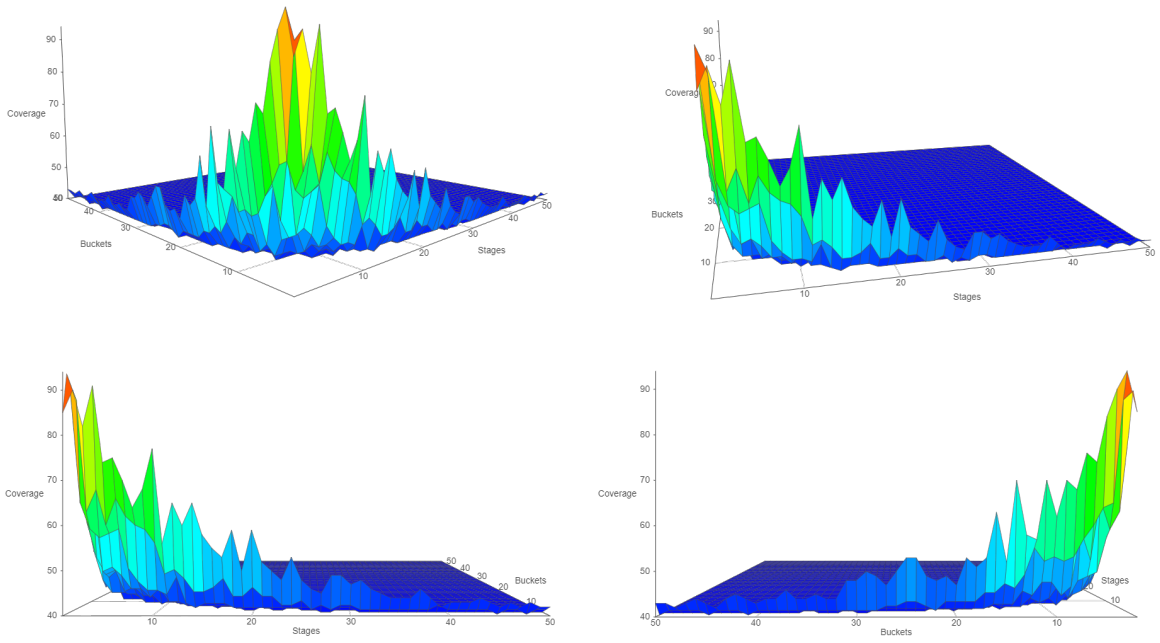


Figure 4.10: Coverage of DBLP-ACM Dataset (Single Feature Hashing)

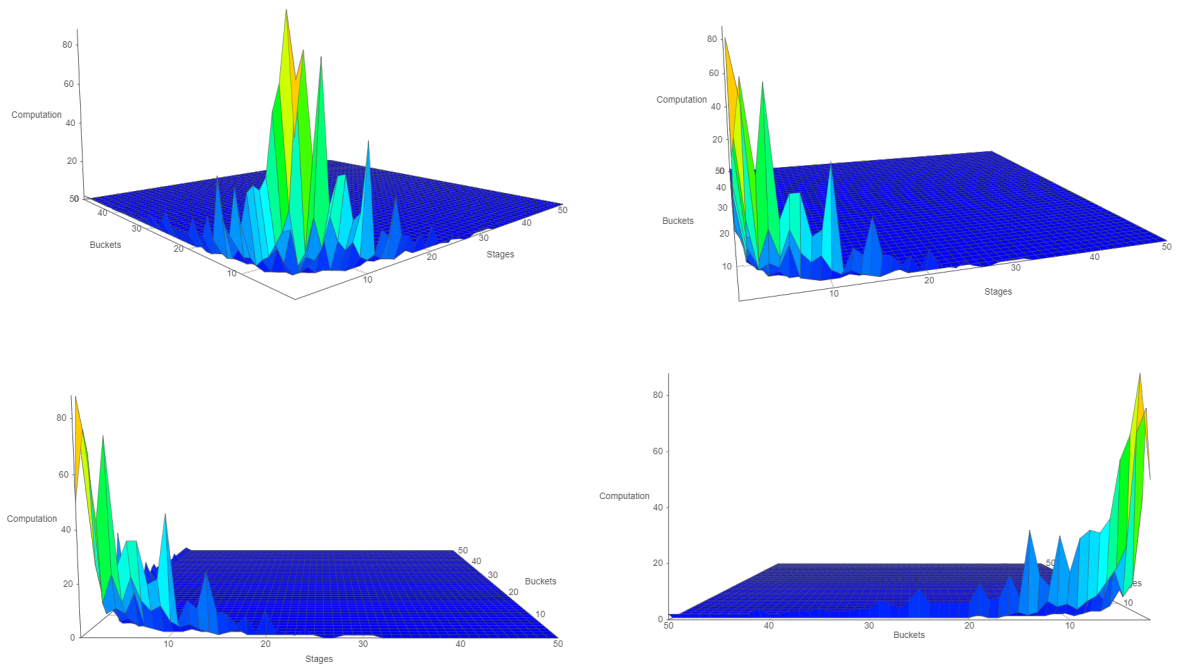


Figure 4.11: Computation of DBLP-ACM Dataset (Single Feature Hashing)

For our experimental evaluation, we consider the trade-off between these Coverage and Computation values and thus Figure 4.12 provides for this trade-off analysis in a Pareto Front plot. To analyse further, we selectively label the points of interest with Computation and Coverage > = 50, as shown in Figure 4.13. Results show, that when contrasted with the Single Feature hashing of the Amazon-Google dataset, the DBLP-ACM reaches a wider scope of possible trade-off solutions with high coverage. Unfortunately, these solutions still incur in a high computation, having code lengths between 1-4.

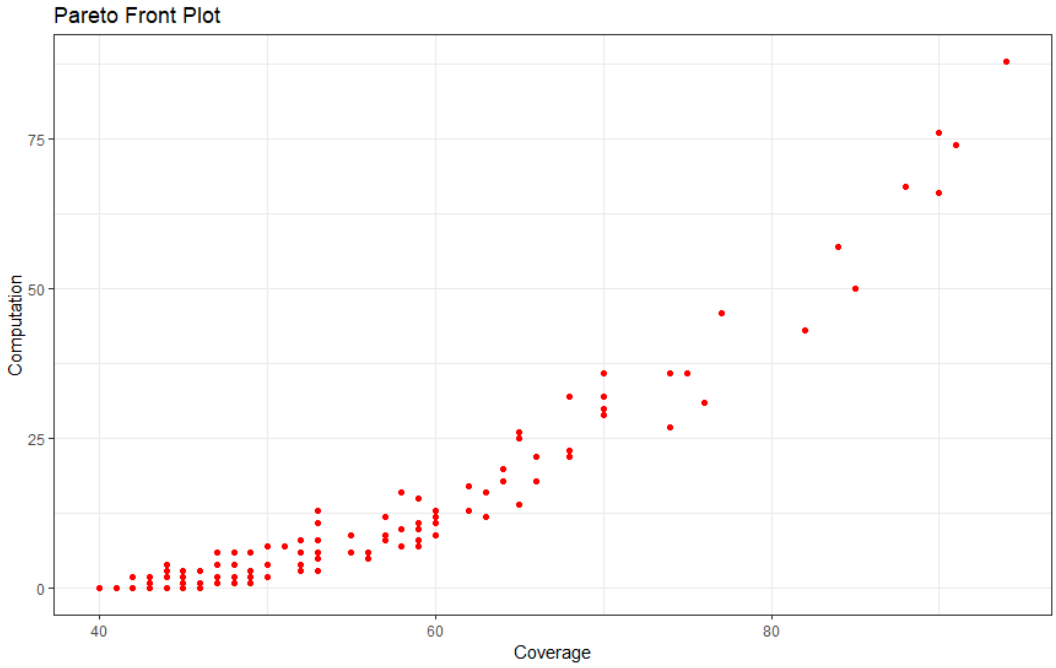


Figure 4.12: Pareto Front Plot of DBLP-ACM Dataset (Single Feature Hashing)

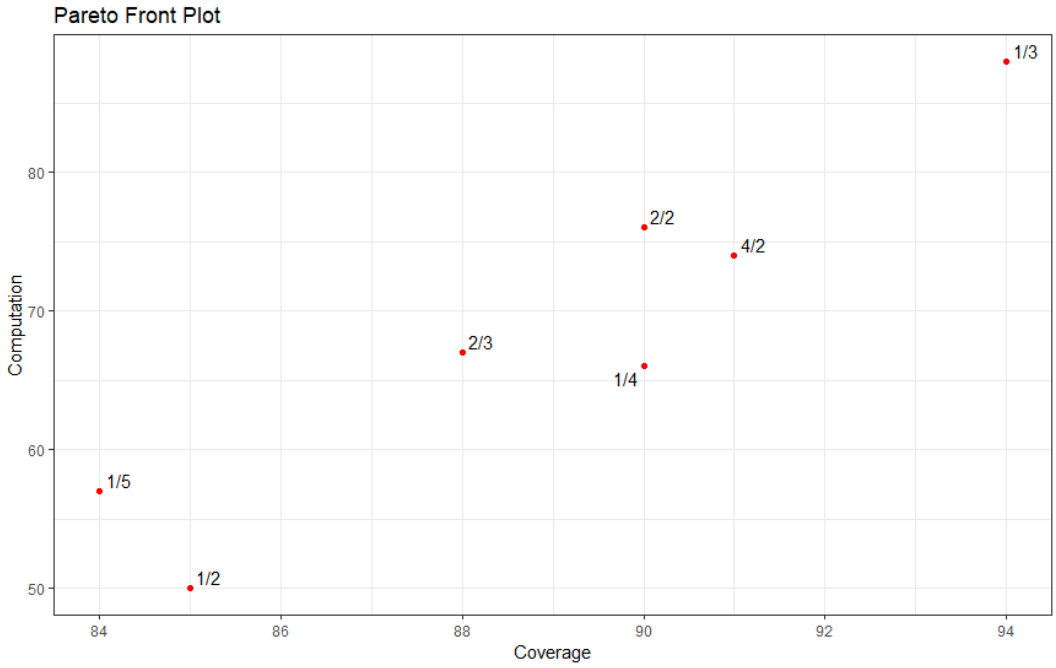


Figure 4.13: Pareto Front Plot of DBLP-ACM Dataset (Selective Labelling Single Feature Hashing)

– **All Features**

Figure 4.14 and Figure 4.15 depict the Coverage and Computation for All Features hashing on DBLP-ACM dataset. The upper flat layer demonstrates a large number of stages & buckets combinations having coverage or computation to be 100%. These observations suggest that the technique works well, in terms of maximizing coverage for this dataset and All Features hashing. The plots also show a lot of variability, suggesting trade-off opportunities, though it is unclear if these will be able to achieve high coverage and low computation. For that we defer to our trade-off analysis.

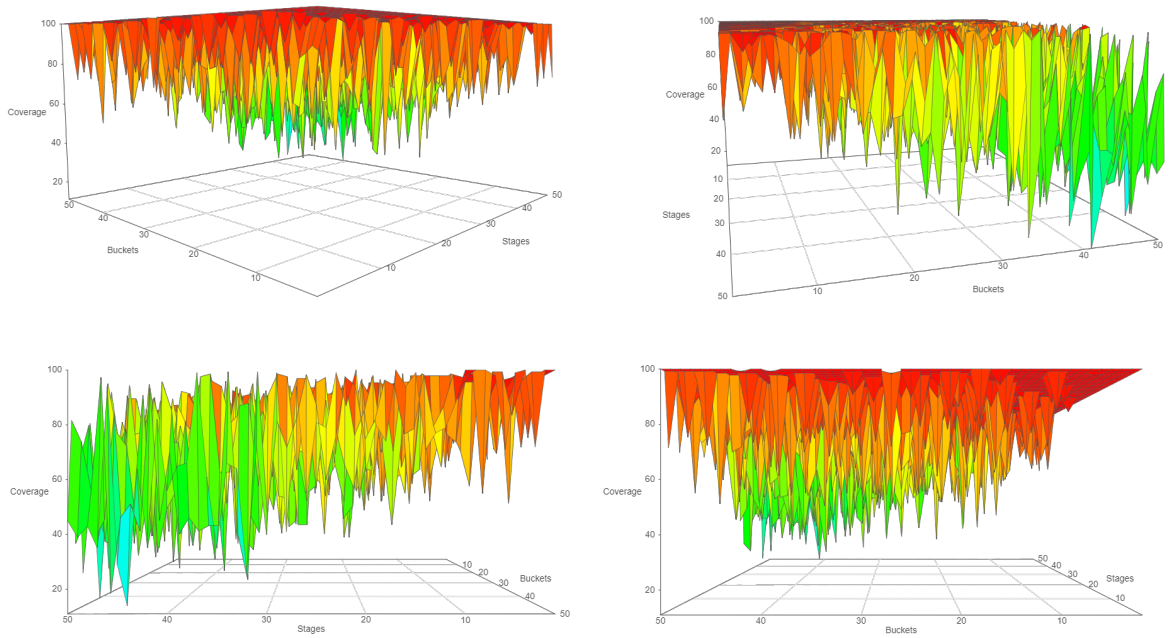


Figure 4.14: Coverage of DBLP-ACM Dataset (All Features Hashing)

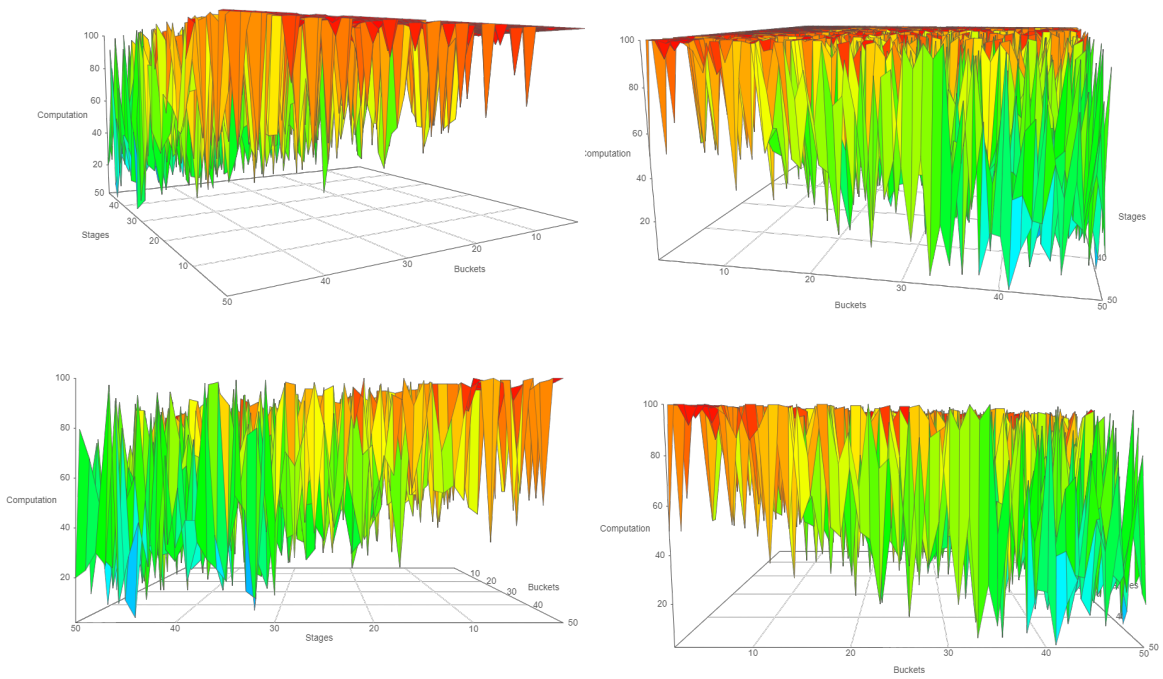


Figure 4.15: Computation of DBLP-ACM Dataset (All Features Hashing)

For our experimental evaluation, we consider the trade-off between these Coverage and Computation values and thus Figure 4.16 provides for this trade-off analysis in a Pareto Front plot. Due to a large number of points lying in the normally selected Coverage and Computation > = 50 quadrant with a heavy number of points having Coverage and Computation = 100, to analyse further we selectively label the points of interest with Computation and Coverage values lying in between 85 to 95, as shown in Figure 4.17. These results show the large amount of trade-off solutions. In addition, they show that the points having 100% coverage, unfortunately remain incurring in high computation costs, with code lengths of around 1-4. The space of possible solutions in the Pareto frontier shows a large variation in code lengths.

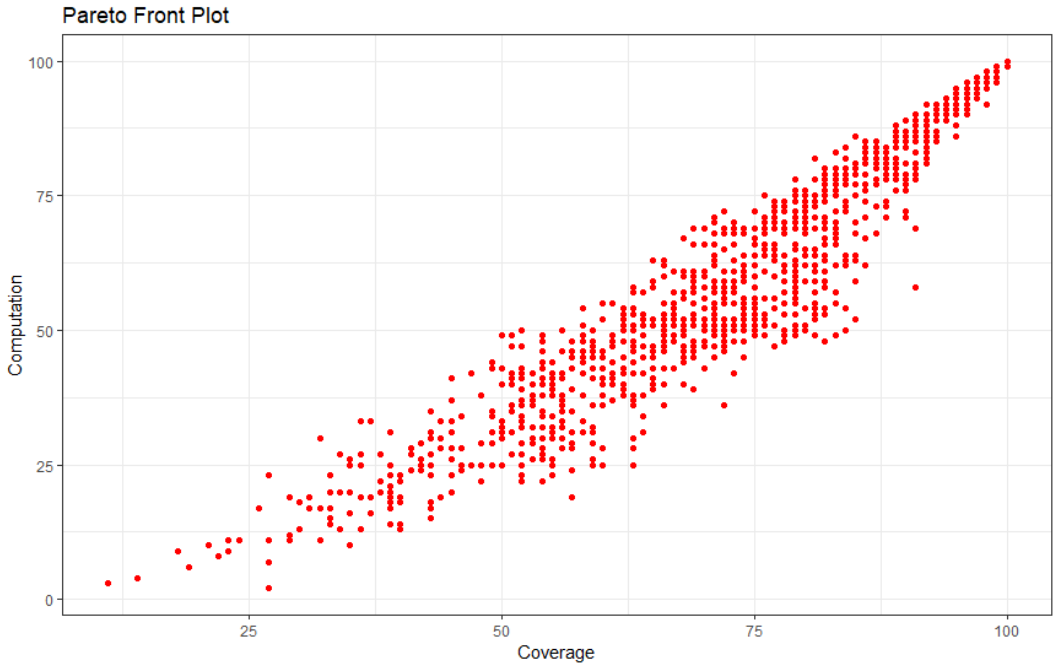


Figure 4.16: Pareto Front Plot of DBLP-ACM Dataset (All Features Hashing)

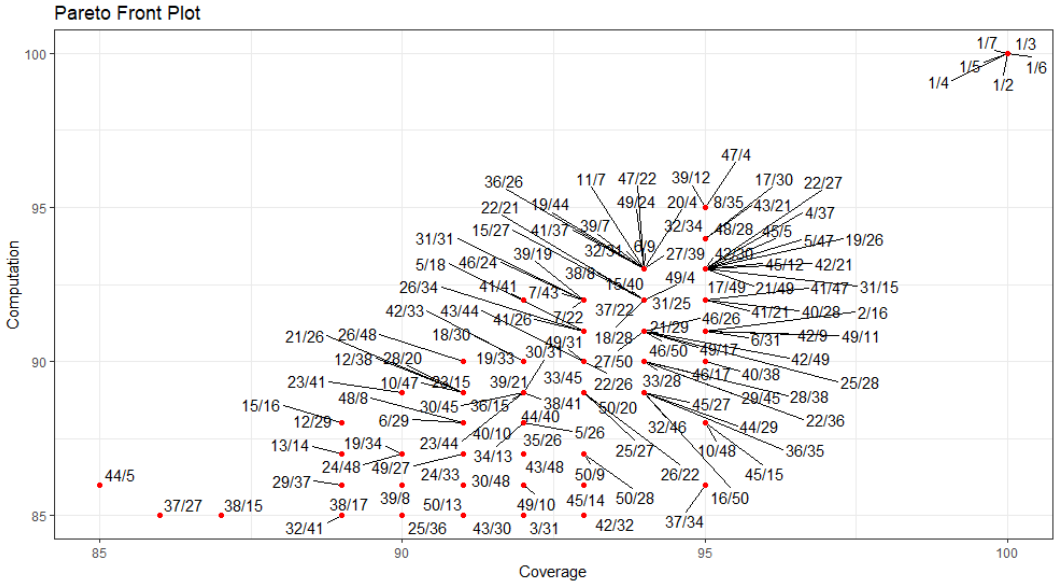


Figure 4.17: Pareto Front Plot of DBLP-ACM Dataset (Selective Labelling All Features Hashing)

- **Walmart-Amazon Dataset** The evaluation results on Walmart-Amazon Dataset are as below:

  - **Single Feature**

Figure 4.18 and Figure 4.19 depict the Coverage and Computation for Single Feature hashing on Walmart-Amazon dataset. The observed results are similar to the Amazon-Google dataset for the Single Feature hashing. A coverage of 100% is not reached, and the solutions with high coverage only seem to occur at cases where the buckets and stages are small. Computation is overall low, but it peaks for the cases where the coverage is high.

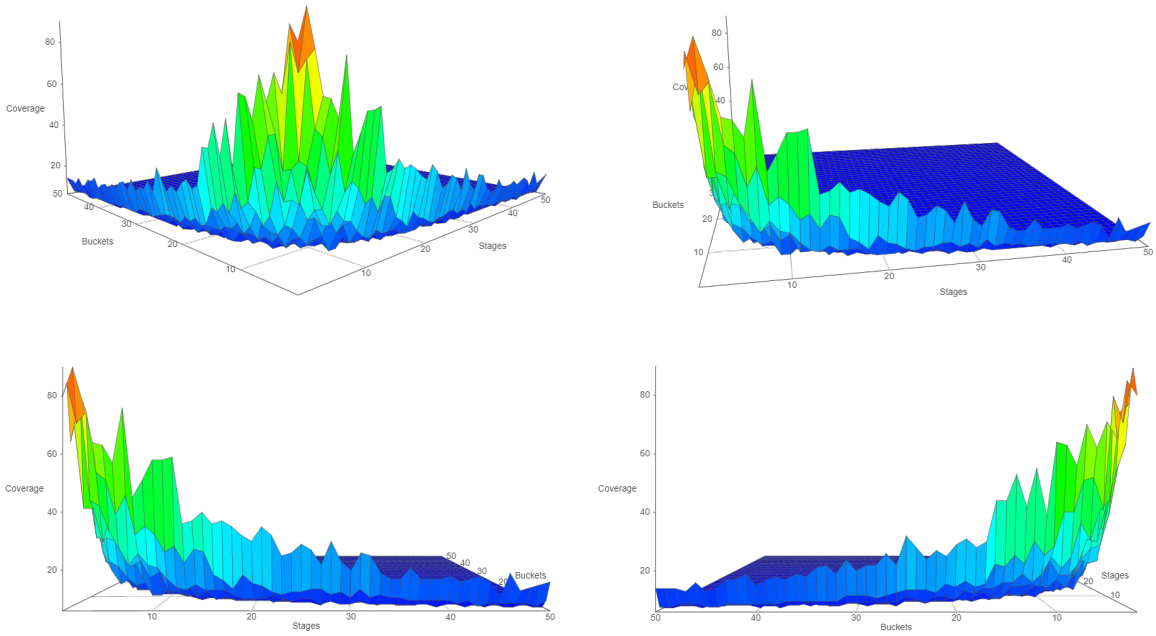


Figure 4.18: Coverage of Walmart-Amazon Dataset (Single Feature Hashing)

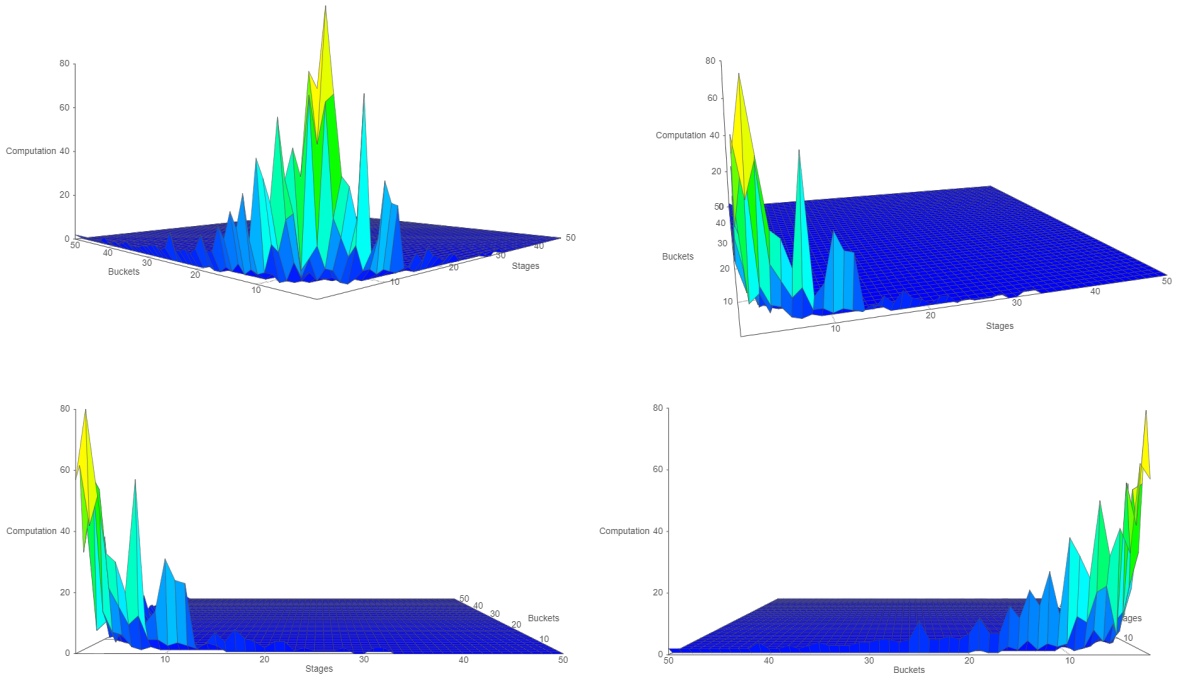


Figure 4.19: Computation of Walmart-Amazon Dataset (Single Feature Hashing)

For our experimental evaluation, we consider the trade-off between these Coverage and Computation values and thus Figure 4.20 provides for this trade-off analysis in a Pareto Front plot. To analyse further, we selectively label the points of interest with Computation and Coverage > = 50, as shown in Figure 4.21. We can observe a moderate amount of solutions, with code lengths between 1-7.

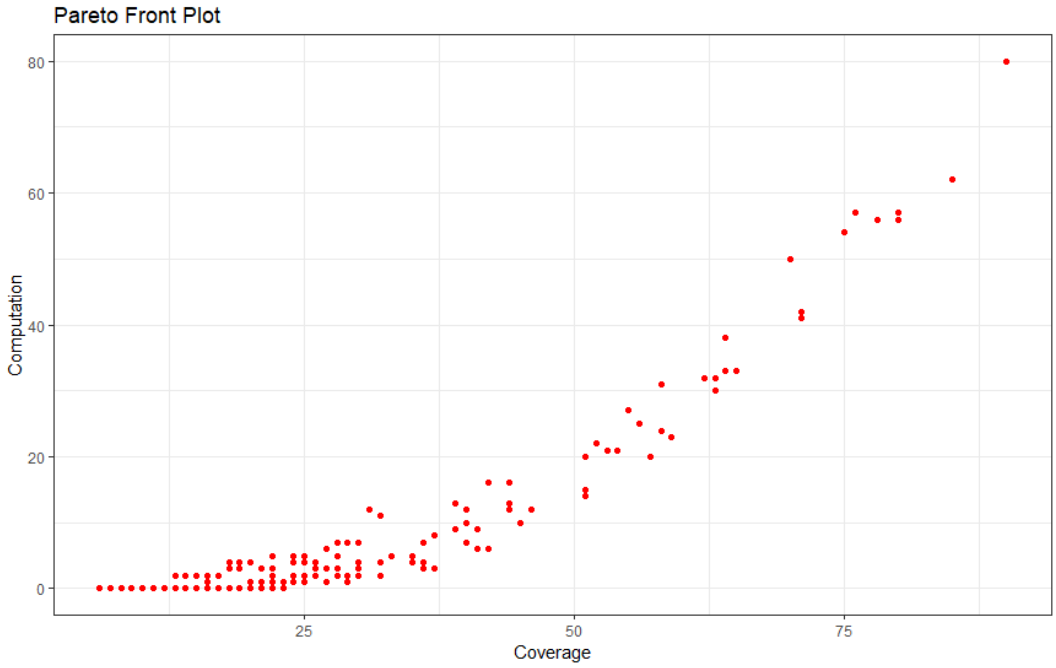


Figure 4.20: Pareto Front Plot of Walmart-Amazon Dataset (Single Feature Hashing)

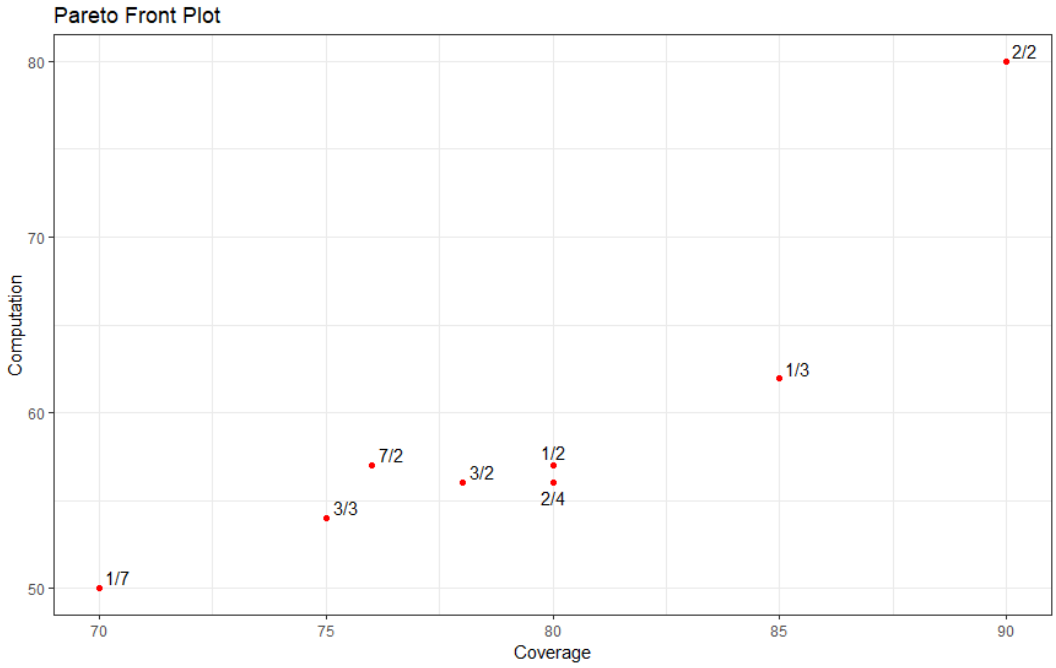


Figure 4.21: Pareto Front Plot of Walmart-Amazon Dataset (Selective Labelling Single Feature Hashing)

– **All Features**

Figure 4.22 and Figure 4.23 depict the Coverage and Computation for All Features hashing on Walmart-Amazon dataset. Results are similar to the Amazon-Google dataset for the case of All Features hashing. We observe an increase in coverage, with more trade-off solutions present, when compared to the Single Feature hashing. We also observe that a coverage close to 100% is reported. Computation shows some variation, without a marked deterioration when contrasted to the Single Feature hashing.

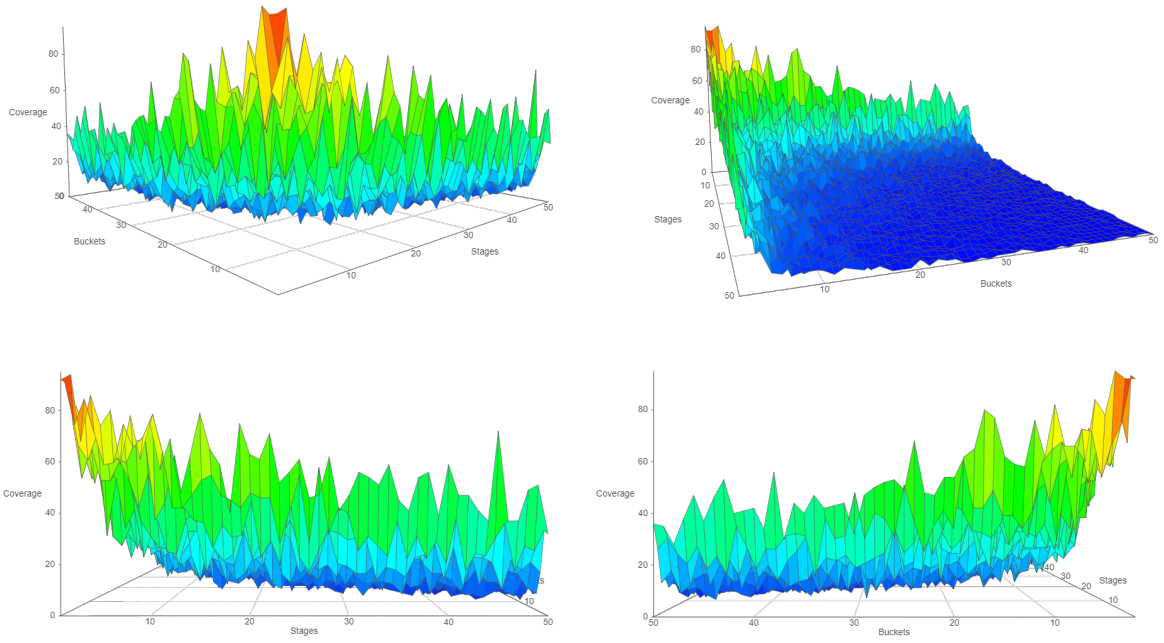


Figure 4.22: Coverage of Walmart-Amazon Dataset (All Features Hashing)

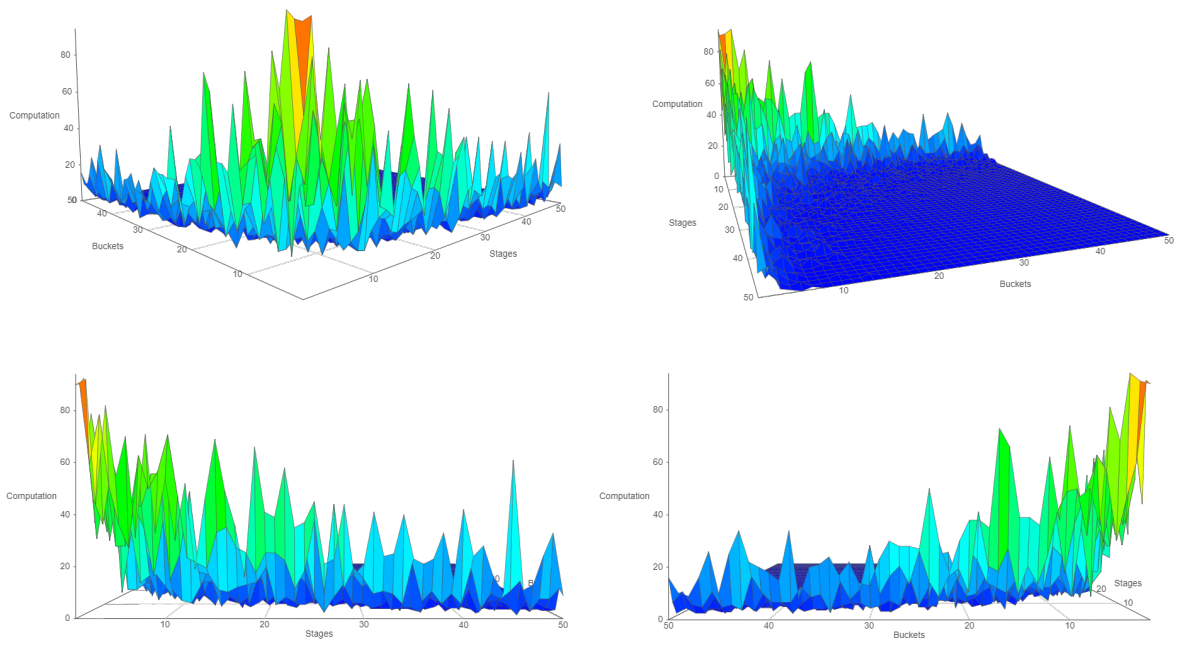


Figure 4.23: Computation of Walmart-Amazon Dataset (All Features Hashing)

For our experimental evaluation, we consider the trade-off between these Coverage and Computation values and thus Figure 4.24 provides for this the trade-off analysis between the Coverage and Computation values in a Pareto Front plot for All Features hashing on Walmart-Amazon dataset. To analyse further, we selectively label the points of interest with Computation and Coverage > = 50, as shown in Figure 4.25. Results show a large amount of solutions being offered, with code lengths between 1-45.

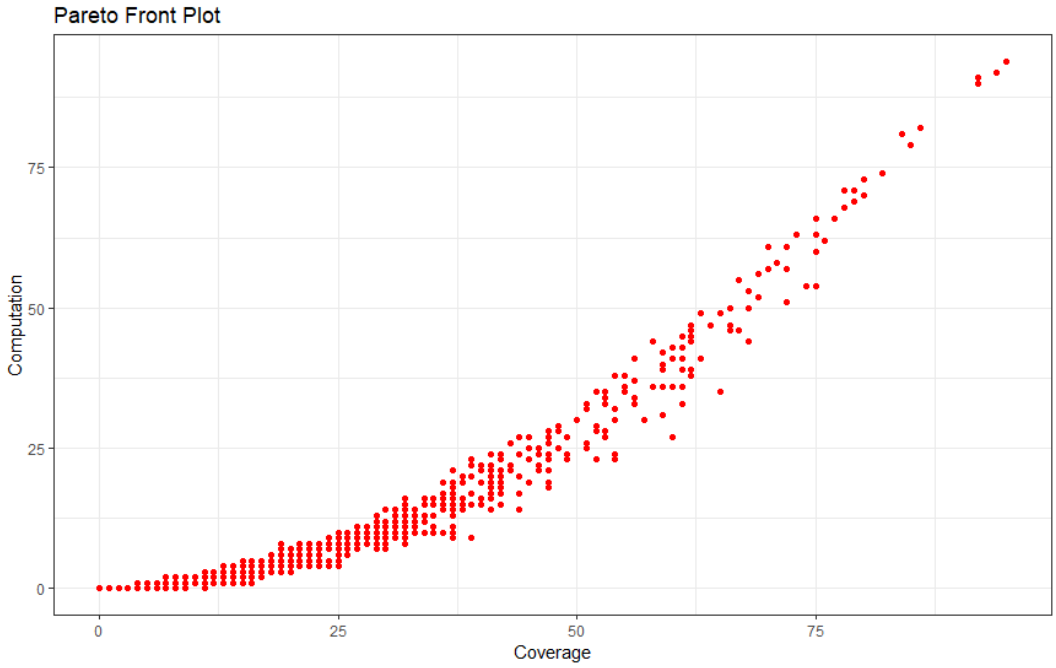


Figure 4.24: Pareto Front Plot of Walmart-Amazon Dataset (All Features Hashing)

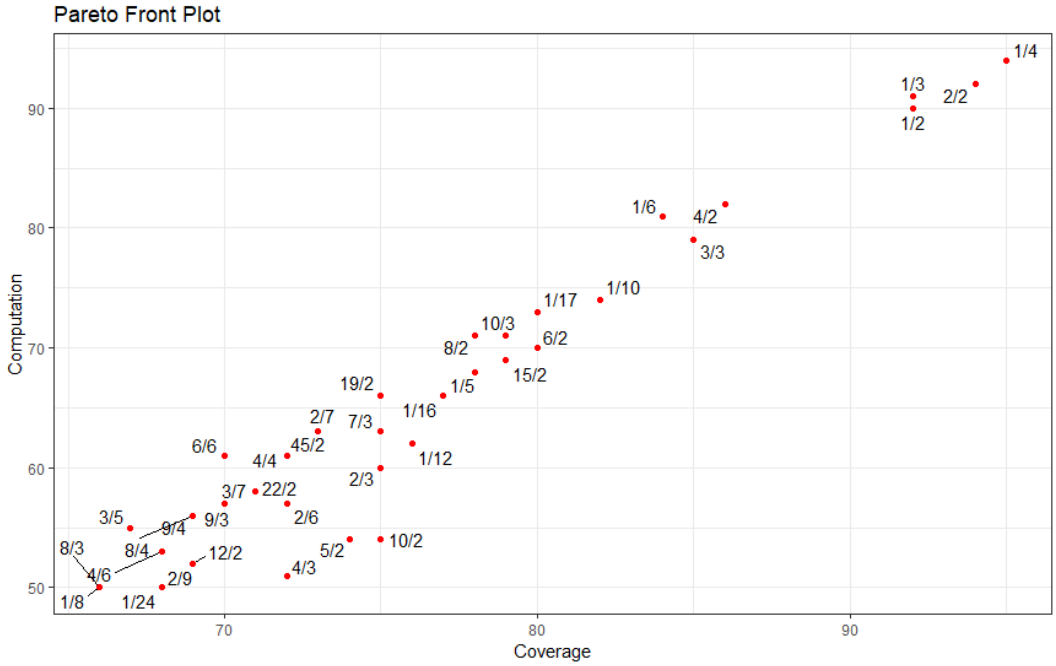


Figure 4.25: Pareto Front Plot of Walmart-Amazon Dataset (Selective Labelling All Features Hashing)

- **Classification Accuracy for the Evaluation of the Task of Supervised Entity Resolution**

Table 4.1 depicts the classification accuracy in the form of F1 score on all datasets for the Single Feature hashing on the *Left* and for All Features hashing on the *Right* on a carefully selected sample of Stages & Buckets combinations from the corresponding Pareto Frontier solutions being representative of the selected solution space. In doing so, we evaluate the performance of LSH - Super-Bit for the task of supervised entity resolution, using XGBoost as a classifier, as described in Chapter 2 Section 2.5.2.3. Further, the combinations marked in **bold** denote the best-performing ones from the set for the respective datasets. Additionally, evaluating and comparing the classification performance of the datasets for Super-Bit hashing technique, results for DBLP-ACM dataset outperform the two other datasets for both Single and All Features hashing as depicted in Figure 4.26. These results for DBLP-ACM are achieved in cases where the code lengths are large and where they are small, suggesting that there is a limited impact of blocking on the accuracy. For the other datasets, we find the best results for All Features hashing, without a clearly defined relation to the code length (i.e., though the code length seems to help, not always the largest code length leads to the best score).

Stages	Buckets	F1 Score
Amazon-Google Dataset		
1	2	0.304273
2	2	0.301369
1	**4**	**0.333333**
DBLP-ACM Dataset		
1	**3**	**0.947791**
4	2	0.940098
1	4	0.941422
1	2	0.940677
Walmart-Amazon Dataset		
2	2	0.177285
1	**3**	**0.242424**
2	4	0.165745
1	7	0.237196

Stages	Buckets	F1 Score
Amazon-Google Dataset		
1	2	0.374367
1	3	0.309433
2	**13**	**0.385297**
5	9	0.347270
DBLP-ACM Dataset		
1	2	0.932964
1	4	0.939808
37	34	0.935261
42	**32**	**0.940267**
Walmart-Amazon Dataset		
1	4	0.235616
10	**2**	**0.263440**
4	3	0.258575
4	6	0.225250

Table 4.1: F1 Scores for all Datasets reported for Single Feature hashing on the *Left* and for All Features hashing on the *Right*

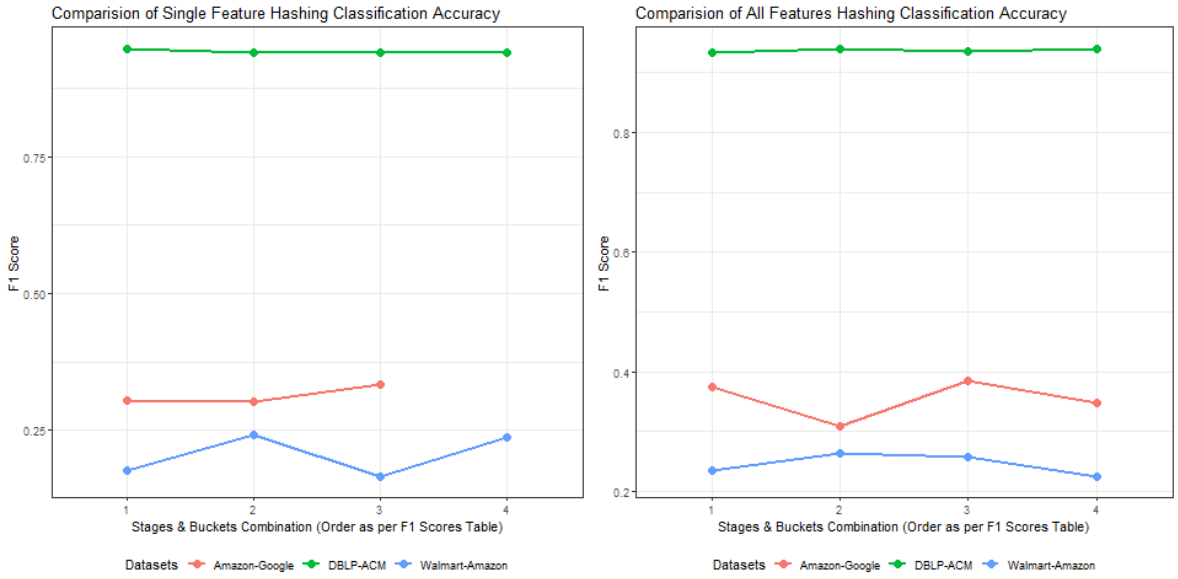


Figure 4.26: Comparison of Classification Accuracy on the Datasets

Answering RQ$_1$, we determine and visualize the best possible values for Coverage and Block Distribution (Computation) for each of the three input dataset pairs namely, Amazon-Google, DBLP-ACM and Walmart-Amazon for Single Feature and All Features high-dimensional hashing using LSH - Super-Bit. Furthermore building upon these base measures, we also determine and visualize the trade-off values for Coverage and Computation depicting the optimal Pareto Frontier solutions for different Stages & Buckets combinations for this hashing technique. Additionally, we also measure the performance of LSH - Super-Bit for supervised entity resolution in terms of classification accuracy achieved after using the hashing technique for blocking.

4.4 Summary

Summarizing, we experimentally evaluate the performance of LSH using Super-Bit for word embeddings on three dataset pairs namely, Amazon-Google, DBLP-ACM and Walmart-Amazon for *Single Feature* and *All Features* hashing. We use three metrics including two user-defined metrics namely *Coverage* measuring the number of correct similar mappings to the same bucket by LSH - Super-Bit based on the Ground Truth dataset acting as a benchmark for similarity & *Computation* capturing the block distribution by LSH - Super-Bit and *F1 score* measuring the classification accuracy for similarity search using LSH - Super-Bit. Additionally, we depict the Coverage and Computation for each of the datasets through intuitive 3D plots visualizing each of their relationship to the input Stages and Buckets for LSH - Super-Bit. Furthermore, we consider the trade-off between Coverage and Computation for our experimental evaluation and visualize it through a Pareto Front plot depicting the Pareto Frontier optimal solutions for different Stages & Buckets combinations. We find that All Features hashing provides more solutions in the trade-off space, for all datasets, while

Single Feature hashing seems to provide only solutions where high computation is required for high coverage. We also find a notably high number of solutions for DBLP-ACM, which are able to reach high coverage and let users decide on what amount of computation to configure.

We further utilize four of these carefully selected combinations to test for their classification accuracy based on the similarity measure *Cosine Similarity* between all the related features of the hashed vectors. In doing so, we evaluate the performance of LSH - Super-Bit for the task of supervised entity resolution. Consequently, the performance of LSH - Super-Bit can be tuned accordingly by adjusting the Stages & Buckets as suggested by the high classification accuracy, i.e., F1 score of a particular combination on the dataset. Furthermore observing upon the classification accuracy, DBLP-ACM dataset outputs the higher F1 score being almost close to 1 for all the selected Pareto Frontier solutions irrespective of Single Feature or All Features hashing; these results suggest a limited impact of blocking for this dataset. Regarding the remaining two dataset pairs namely Amazon-Google and Walmart-Amazon, the output F1 scores did not seem to perform significantly well, but they increase slightly when we utilize All Features hashing as compared to Single Feature hashing on the datasets. To further effectively measure, compare and draw solid conclusions regarding the performance of LSH - Super-Bit hashing technique on high-dimensional data, we consider and evaluate another high-dimensional hashing technique *Learning To Hash - Deep Hashing* being data-dependent utilizing the same input datasets in the next Chapter 5.

[1]https://xgboost.readthedocs.io/en/latest/index.html

5. Learning To Hash

In this chapter, we focus on a data-dependent high-dimensional hashing technique, which reduces the dimensions of the high-dimensional input data items converting them into low-dimensional versions while preserving the relative similarity between them. Specifically, we focus on Learning To Hash (L2H) and experimentally evaluate the performance of a Deep Hashing learning method, as described in Section 3.3.

This chapter is structured as follows:

- Section 5.1 recapitulates the research question of focus for this chapter.
- Section 5.2 presents an overview of our data pipeline for L2H.
- Section 5.3 discusses and presents our evaluation results for L2H technique.
- Section 5.4 presents an effective summarization of the important aspects of this chapter.

5.1 Research Question

Recapitulating the research questions we established for this work, in this chapter, we will be addressing the following specific research question:

- **RQ**$_2$: How does Learning To Hash compare, with regards to blocking and coverage, to the standard high-dimensional hashing technique, Locality Sensitive Hashing? How does it compare in the task of supervised entity resolution?

  Relevance Measuring the performance of a L2H technique in comparison to a representative LSH technique helps to evaluate the difference between data-dependent and data-independent hashing techniques on embedded data, providing insights on the strengths and weaknesses of the approaches.

Description This research question deals with the experimental evaluation of L2H (Deep Hashing) using two user-defined metrics namely, *Coverage*, *Computation*. For the task of supervised entity resolution, we also employ the *F1 score*. These evaluation metrics are elaborated upon in Chapter 3 Section 3.4.4. We also depict the coverage variations for neighboring matches demonstrating the number of bit changes required in the original generated hash code to achieve the optimal coverage. Additionally, we evaluate the reported coverage and computation values with a baseline Brute-Force approach.

5.2 Data Pipeline

In our study on high-dimensional hashing using L2H, we primarily choose three publicly available structured dataset pairs as elaborated upon in Chapter 3 Section 3.4.2.

5.2.1 Pre-Processing and Vectorization

The pre-processing and vectorization carried out on these input datasets is as elaborated in Chapter 3 Section 3.4.3.

5.2.2 Learning To Hash Technique

Learning To Hash (L2H)[1] [18, 108] as discussed in Chapter 2 Section 2.5.2.2 are a set of data-dependent hashing methods which aim to learn a compact and similarity-preserving bitwise representation with shorter hash codes in such a way that similar inputs are mapped to nearby binary hash codes. Prominently, deep hashing methods based on supervised learning which are capable of performing feature learning and hash code learning simultaneously, have demonstrated superior performance over traditional hashing methods with application to image retrieval tasks [10]. Consequently, in our thesis, we study and evaluate deep hashing on the word embedding data. The different state-of-art in the area of deep hashing are elaborated in Chapter 2 Section 2.5.2.2. Specifically, we experiment and evaluate with our proposed Deep Hashing approach *Deep Hash Neural Net (DHNN)* as elaborated in Chapter 3 Section 3.3.

5.2.3 Evaluation

The pre-selected evaluation parameters for L2H include, *Code Length* denoting the bitcode length of the generated hash code. We test the performance of this hashing technique utilizing *Single Feature Hashing* and *All Features Hashing* which differs in the embedding dimension given as an input to the deep hashing network i.e., 300 for Single Feature Hashing and 901 for All Features Hashing. The Single Feature hashing attributes selected for the dataset include *Product Name* for Amazon-Google, *Paper Name* for DBLP-ACM, and *Title* for Walmart-Amazon. For each of these two variations, the optimal Code Length which output better hashing performance is discovered based on two user-defined metrics *Coverage* and *Computation* and further evaluated for classification accuracy based on the metric *F1 score*

(as discussed in Chapter 3 Section 3.4.4). Consequently, we focus on the trade-off between these two values, and hence, we measure this through a trade-off analysis plot known as the *Pareto Front*. Furthermore, to test for classification accuracy for similarity search, we utilize the popular *Cosine Similarity* measure [217–219] between all the related features of the hashed vectors for different Code Lengths. We then utilize eXtreme Gradient Boosting or XGBoost[2][220] to test for this classification accuracy based on F1 score and eventually report these results.

Consequently, the evaluation criteria are as summarized below:

- **Benchmark** A benchmark indicator of similarity as the input Ground Truth dataset consisting of ID-ID pairs mapping providing the supervised label information.
- **Baseline** Brute-Force Approach which exhaustively maps the similar key pairs to similar randomly-assigned hash codes attempting to achieve the highest coverage and the lowest computation values.
- **Similarity Calculation** Similarity calculation to calculate similarity scores between the hashed vectors with Cosine Similarity.
- **Evaluation Metrics** Evaluation Metrics to assess the performance of our deep hashing technique with Coverage, Computation, F1 score.
- **Tuning Parameters** Code Length.

5.3 Results

In this section, we present and experimentally evaluate the results of our study on the three dataset pairs, thus answering our formulated RQ_2. The results are categorized into two variations as *Single Feature* and *All Features* with Single Feature depicting the hashing results with Single Feature Hashing & corresponding classification results with Single Feature Hashing-All Features Classification and All Features depicting the hashing results with All Features Hashing & classification results with All Features Hashing-All Features Classification. Each category is experimentally evaluated with seven different *Code Lengths* being 12, 16, 24, 32, 48, 64, 128.

There are correspondingly four plots depicted for each dataset pair, Coverage plot, Computation plot, Pareto Front plot and Coverage Variation plot.

- **Amazon-Google Dataset** The evaluation results on Amazon-Google Dataset are as below:
  - **Single Feature**

    Figure 5.1 and Figure 5.2 depict the Coverage and Computation for Single Feature hashing on Amazon-Google dataset.

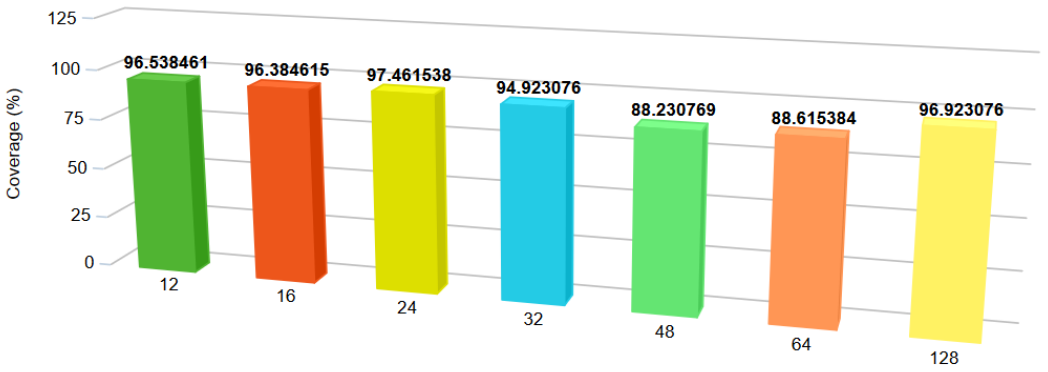


Figure 5.1: Coverage of Amazon-Google Dataset (Single Feature Hashing) for different Code Lengths

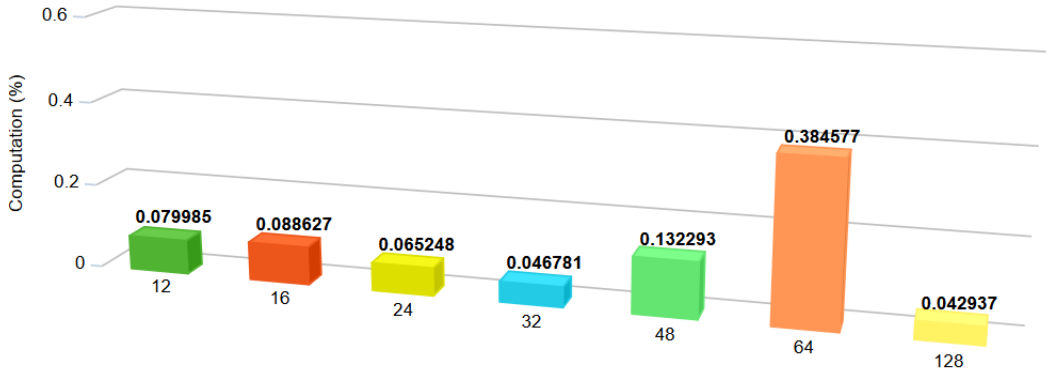


Figure 5.2: Computation of Amazon-Google Dataset (Single Feature Hashing) for different Code Lengths

These results show a marked difference when compared to the Single Feature hashing with LSH. High coverage (although not 100%, all values are higher than 88%) and very low computation are simultaneously achieved. The reduction in computation is especially noteworthy, since values are in less than 0.5% of what the Cartesian product would have been. The maximum coverage is achieved for the 24-bit code length; however, the minimum computation is achieved with the 128-bit code length. Hence for our experimental evaluation, we consider the trade-off between these coverage and computation values and thus Figure 5.3 provides for this trade-off analysis in a Pareto Front plot for the different code lengths. Consequently, these code lengths can be further evaluated based on their classification accuracy.

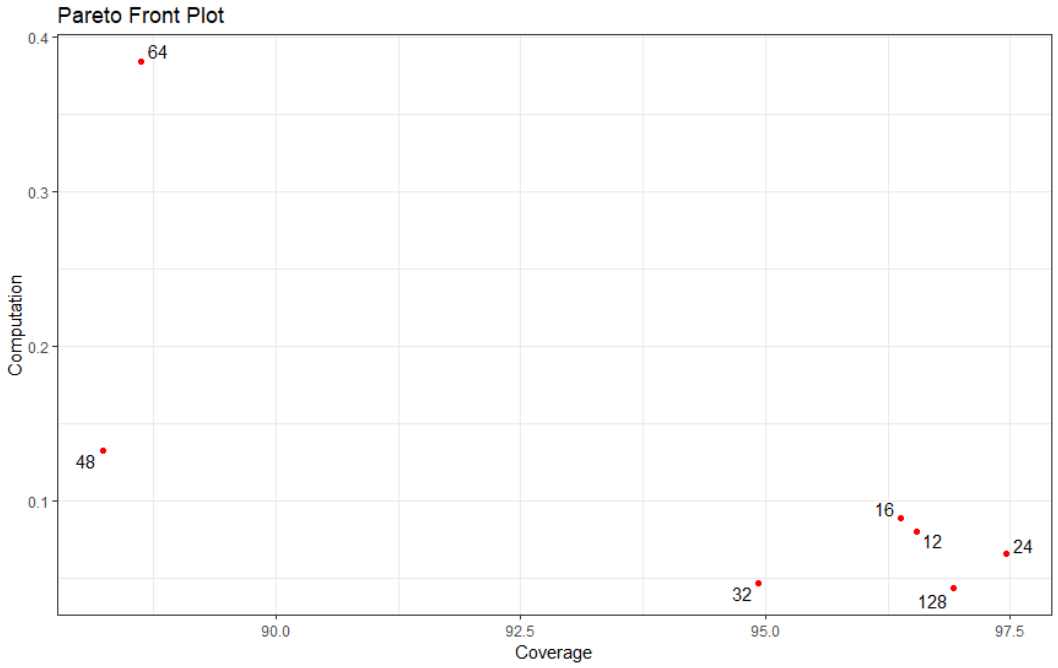


Figure 5.3: Pareto Front Plot of Amazon-Google Dataset (Single Feature Hashing) for different Code Lengths

– **All Features**

Figure 5.4 and Figure 5.5 depict the Coverage and Computation for All Features hashing on Amazon-Google dataset. Results for coverage show some difference when compared to Single Feature hashing, reaching higher and lower values. In terms of computation, we find too, some cases similar to the Single Feature hashing, with very low computation, but we also find cases where learning does not seem to have been successful, producing high computation values. These results might be understandable since the input to the neural network is larger, posing more challenges for the learning. In future research, we will consider whether network improvements or studying training stability aspects might help to better these results.

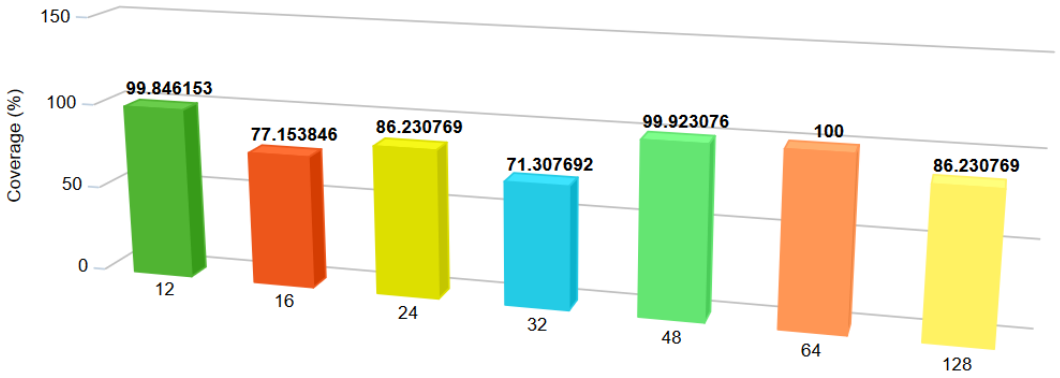


Figure 5.4: Coverage of Amazon-Google Dataset (All Features Hashing) for different Code Lengths

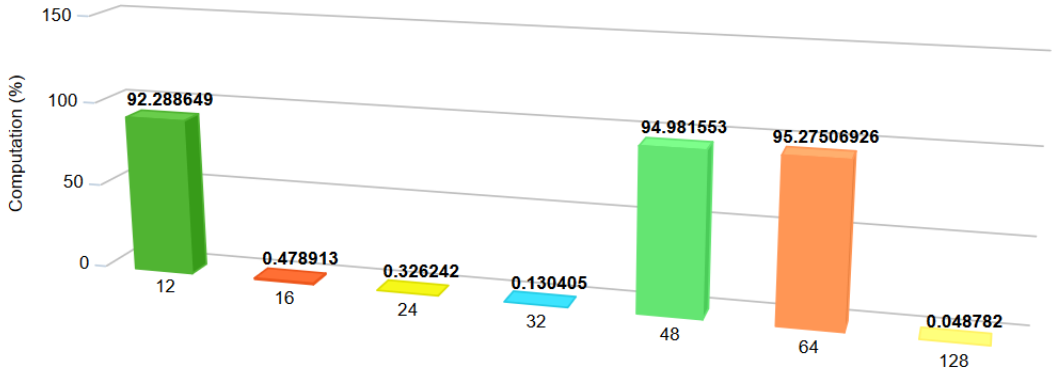


Figure 5.5: Computation of Amazon-Google Dataset (All Features Hashing) for different Code Lengths

The maximum coverage is achieved through the 64-bit code length; however, its computation value is also higher. The minimum computation is achieved through the 128-bit code length. Hence for our experimental evaluation, we consider the trade-off between these coverage and computation values and thus Figure 5.6 provides for this trade-off analysis in a Pareto Front plot for different code lengths. Consequently, these code lengths can be further evaluated based on their classification accuracy. In contrast to all results for the evaluated LSH, we find that solutions cluster either at low or high computation values, with very high coverage achieved at the cost of very high computation. Trade-off solutions reduce 10% from the optimal coverage but have a decrease in computation of almost 100%.

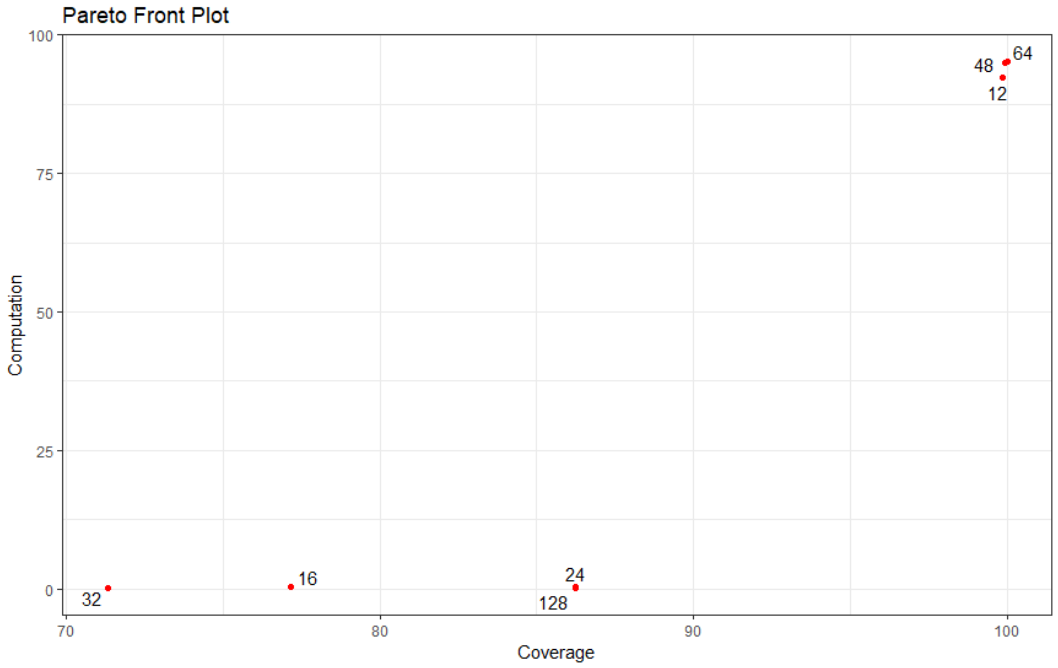


Figure 5.6: Pareto Front Plot of Amazon-Google Dataset (All Features Hashing) for different Code Lengths

- **DBLP-ACM Dataset** The evaluation results on DBLP-ACM Dataset are as below:
  - **Single Feature**

    Figure 5.7 and Figure 5.8 depict the Coverage and Computation for Single Feature hashing on DBLP-ACM dataset for different code lengths. Similar to the results for L2H with Single Feature hashing for the Amazon-Google dataset, we observe a high coverage, while having very low computation values across the board.

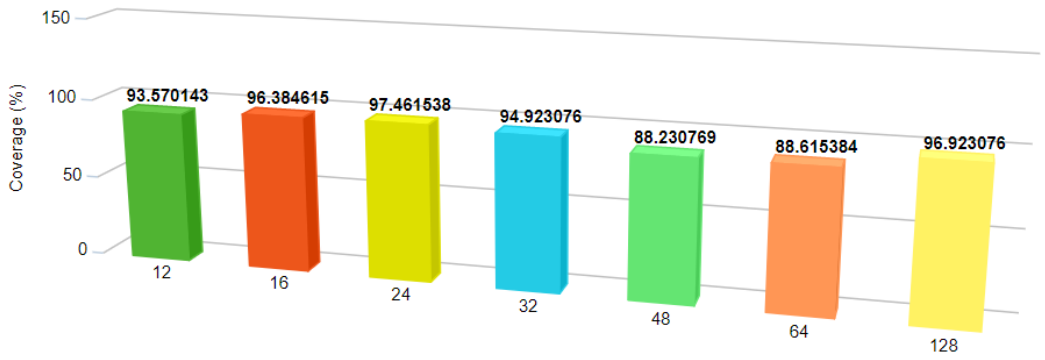


Figure 5.7: Coverage of DBLP-ACM Dataset (Single Feature Hashing) for different Code Lengths

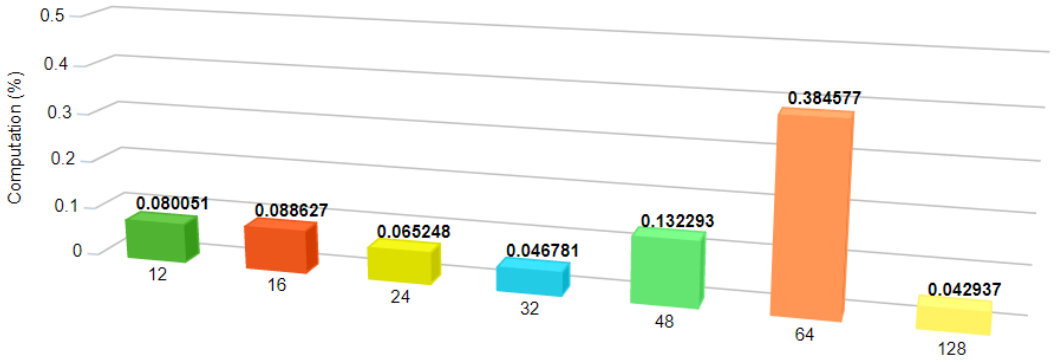


Figure 5.8: Computation of DBLP-ACM Dataset (Single Feature Hashing) for different Code Lengths

The maximum coverage is achieved through the 24-bit code length; however, the minimum computation is achieved through the 128-bit code length. Hence for our experimental evaluation, we consider the trade-off between these coverage and computation values and thus Figure 5.9 provides for this trade-off analysis in a Pareto Front plot for different code lengths. Consequently, these code lengths can be further evaluated based on their classification accuracy.

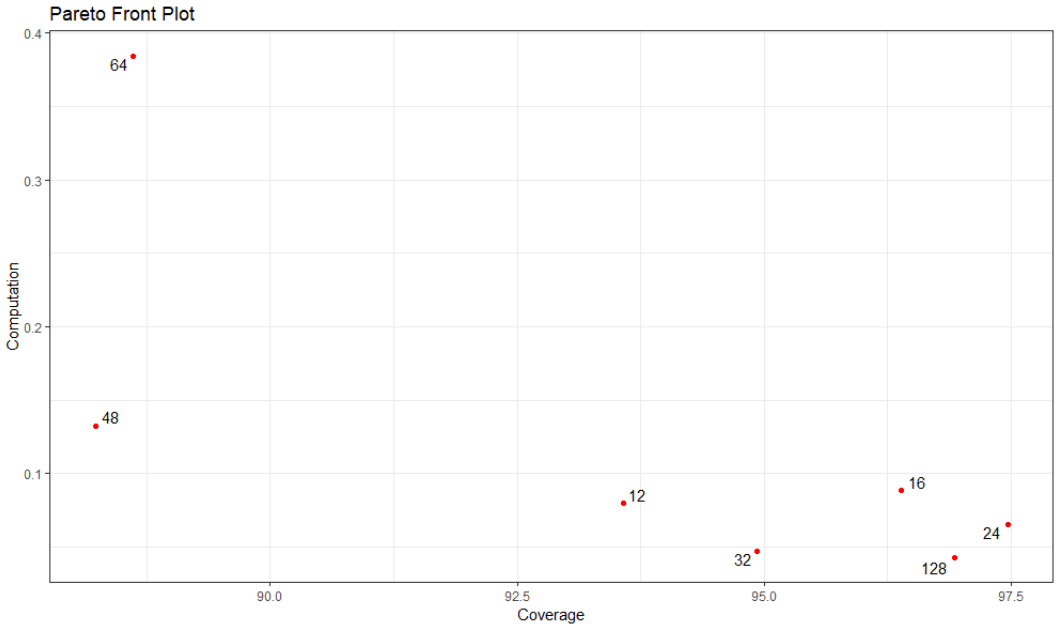


Figure 5.9: Pareto Front Plot of DBLP-ACM Dataset (Single Feature Hashing) for different Code Lengths

– **All Features**

Figure 5.10 and Figure 5.11 depict the Coverage and Computation for All Features hashing on DBLP-ACM dataset for different code lengths. These results are highly

unsuccessful, displaying high coverage and high computation. They suggest the need to carry-out further neural architecture and hyper-parameter tuning to improve the learning for this dataset. However, such a solution was considered to be beyond the scope of our current research.

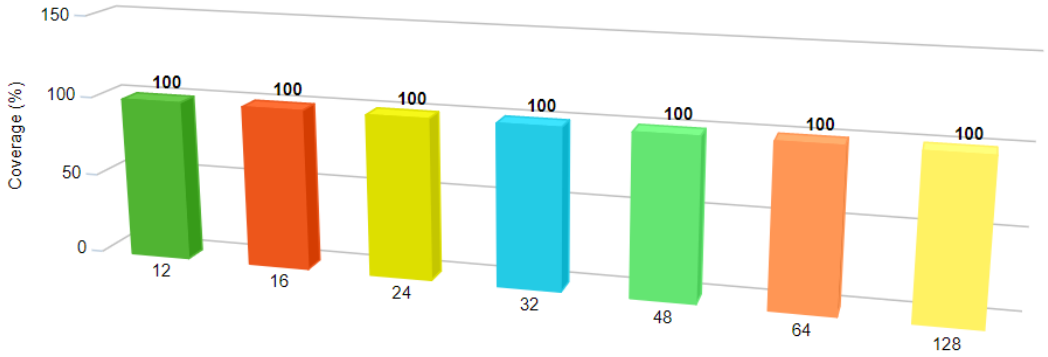


Figure 5.10: Coverage of DBLP-ACM Dataset (All Features Hashing) for different Code Lengths

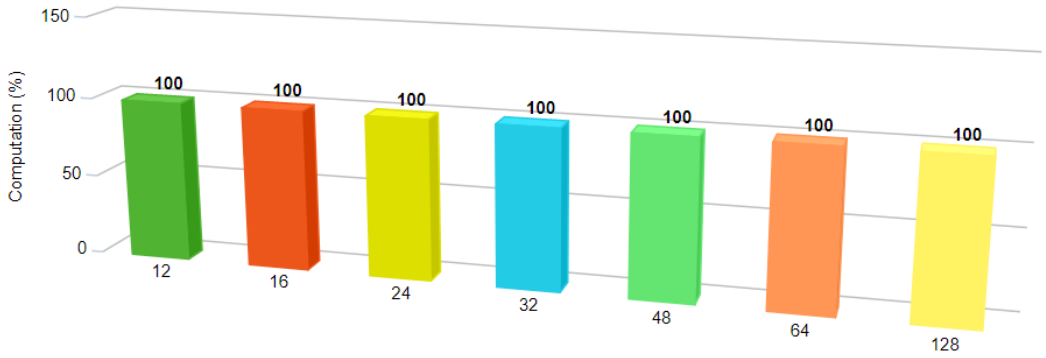


Figure 5.11: Computation of DBLP-ACM Dataset (All Features Hashing) for different Code Lengths

The maximum coverage and minimum computation is achieved through all the bit code lengths. Hence these data points having the same coverage and computation values overlap in the Pareto Front plot Figure 5.12, and thus trade-off analysis for them is not feasible. Thus, we simply pass them on, to evaluate them based on their classification accuracy.

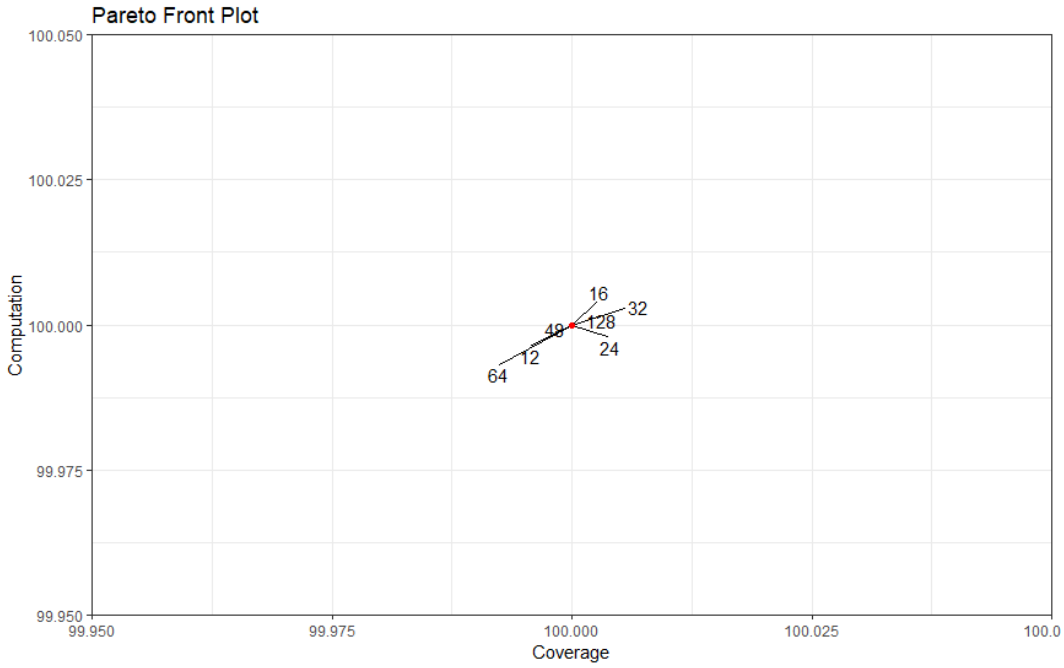


Figure 5.12: Pareto Front Plot of DBLP-ACM Dataset (All Features Hashing) for different Code Lengths

- **Walmart-Amazon Dataset** The evaluation results on Walmart-Amazon Dataset are as below:

  - **Single Feature**

    Figure 5.13 and Figure 5.14 depict the Coverage and Computation for Single Feature hashing on Walmart-Amazon dataset for different code lengths. These results are similar to those of the Amazon-Google dataset for Single Feature hashing with L2H, with high coverage (although not 100%) and low computation values. However, we observe that in some cases, the computation raises higher than 1%. Further studies are required to understand whether different network configurations can improve this, or it is a matter of the stability of the training process.

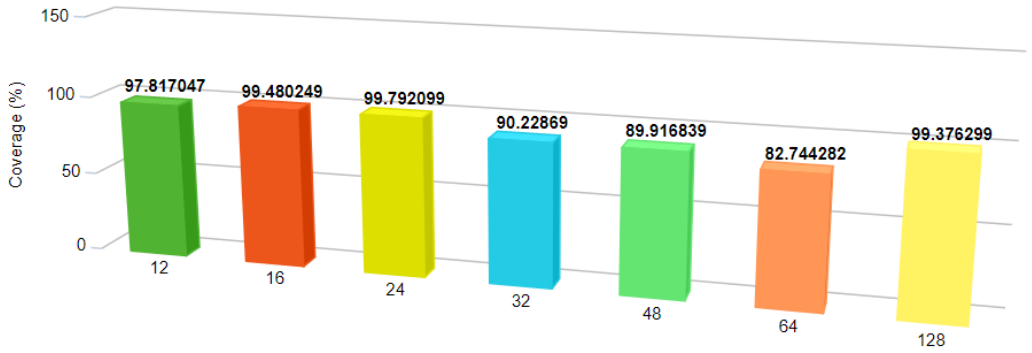


Figure 5.13: Coverage of Walmart-Amazon Dataset (Single Feature Hashing) for different Code Lengths

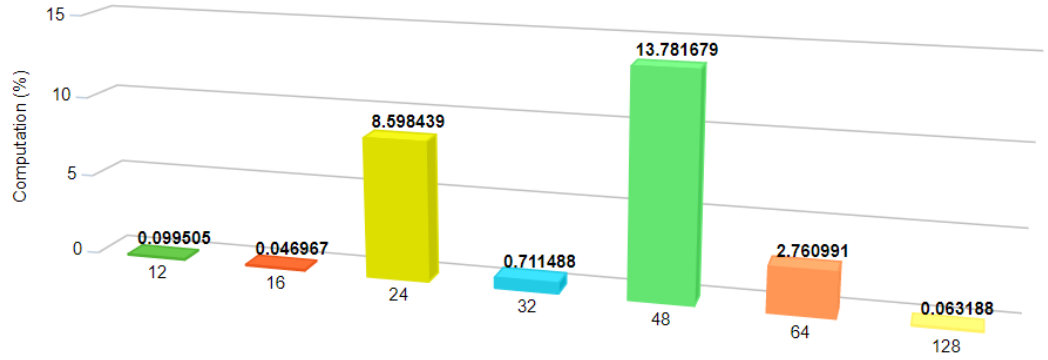


Figure 5.14: Computation of Walmart-Amazon Dataset (Single Feature Hashing) for different Code Lengths

The maximum coverage is achieved through the 24-bit code length; however, the minimum computation is achieved through the 16-bit code length. Hence for our experimental evaluation, we consider the trade-off between these coverage and computation values and thus Figure 5.15 provides for this trade-off analysis in a Pareto Front plot for different code lengths. Consequently, these code lengths can be further evaluated based on their classification accuracy. Results show that there are still several trade-off cases with high coverage and low computation.

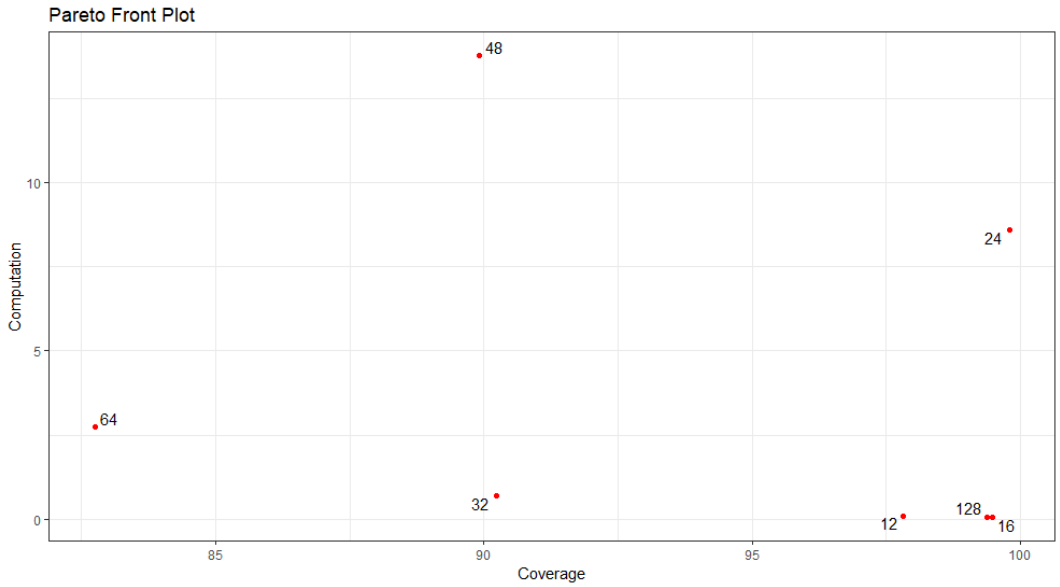


Figure 5.15: Pareto Front Plot of Walmart-Amazon Dataset (Single Feature Hashing) for different Code Lengths

– **All Features**

Figure 5.16 and Figure 5.17 depict the Coverage and Computation for All Features hashing on Walmart-Amazon dataset for different Code Lengths. These results are better than those of All Features hashing with L2H on the Amazon-Google dataset. Both a high coverage (although not 100%) and low computation values are achieved across the board. It should be noted that higher coverage values were reached for this dataset also through the Single Feature hashing.

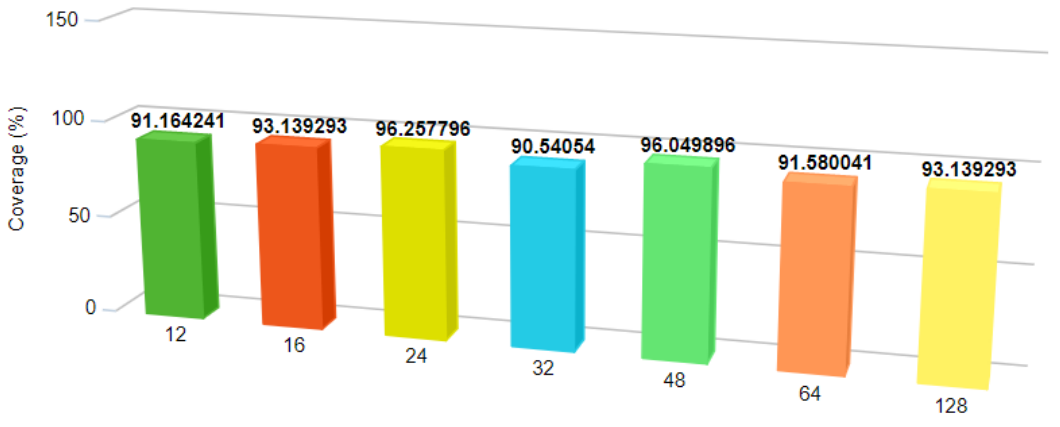


Figure 5.16: Coverage of Walmart-Amazon Dataset (All Features Hashing) for different Code Lengths

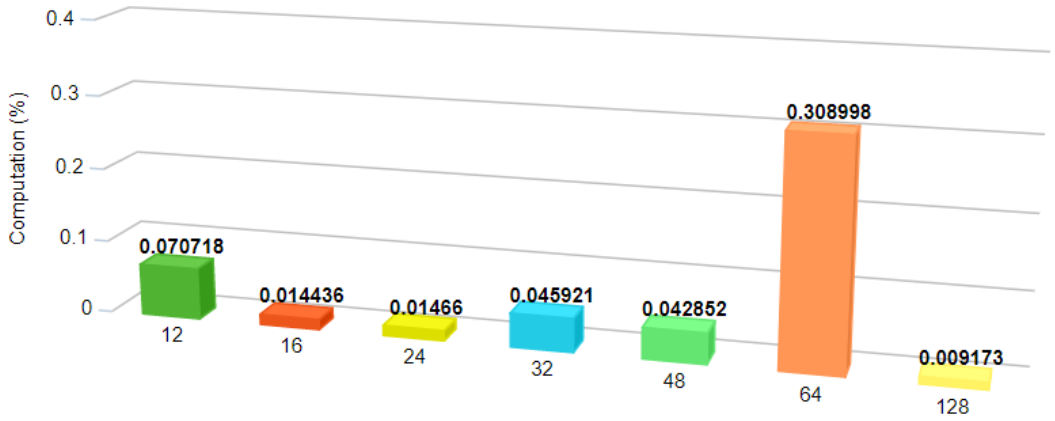


Figure 5.17: Computation of Walmart-Amazon Dataset (All Features Hashing) for different Code Lengths

The maximum coverage is achieved through the 24-bit code length; however, the minimum computation is achieved through the 128-bit code length. Hence for our experimental evaluation, we consider the trade-off between these coverage and computation values and thus Figure 5.18 provides for this the trade-off analysis between the coverage and computation values in a Pareto Front plot for different code lengths. Consequently, these code lengths can be further evaluated based on their classification accuracy.

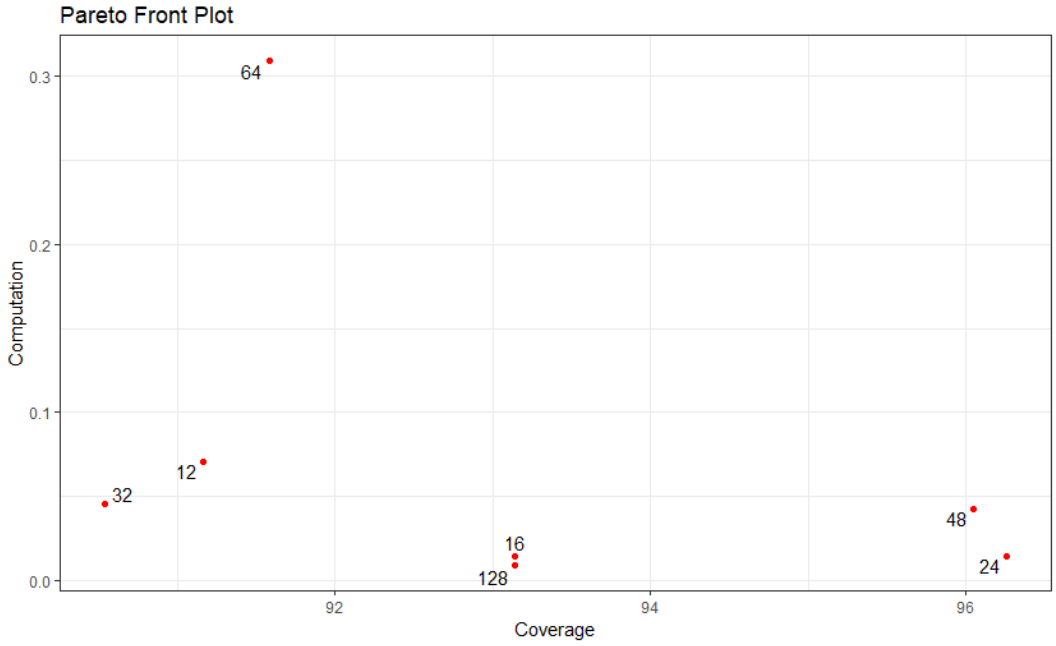


Figure 5.18: Pareto Front Plot of Walmart-Amazon Dataset (All Features Hashing)

- **Coverage Variations for Neighboring Hash Matches**

The following plots depict the coverage variations for the reported neighboring hash matches. 0-bit change interprets the coverage achieved by the generated hash codes. For the remaining values not covered, 1-bit change indicates the coverage achieved through the adjustment of one bit in the generated hash code in addition to the original (0-bit coverage). Likewise, 2, 3....30+ bit changes follow a similar approach. Ideally, this graph should fulfill two properties, depict a constant downward trend and not require too many bit changes to achieve the optimal coverage.

– **Amazon-Google**

Figure 5.19 depicts this plot for the Amazon-Google dataset on Single Feature hashing and Figure 5.20 on All Features hashing. As seen in Figure 5.19, the two properties are satisfied by the code lengths 12, 16, 24, 32, 128 requiring up to 2-bit changes. Code Lengths 48 and 64, however, do not show a constant downward trend and spans up to 10+ bit changes. As seen in Figure 5.20, all the code lengths exhibit a downward trend with code lengths 48, 64 requiring negligible or no bit change with almost or exact 100% coverage. Remaining code lengths require up to 2+ bit changes.

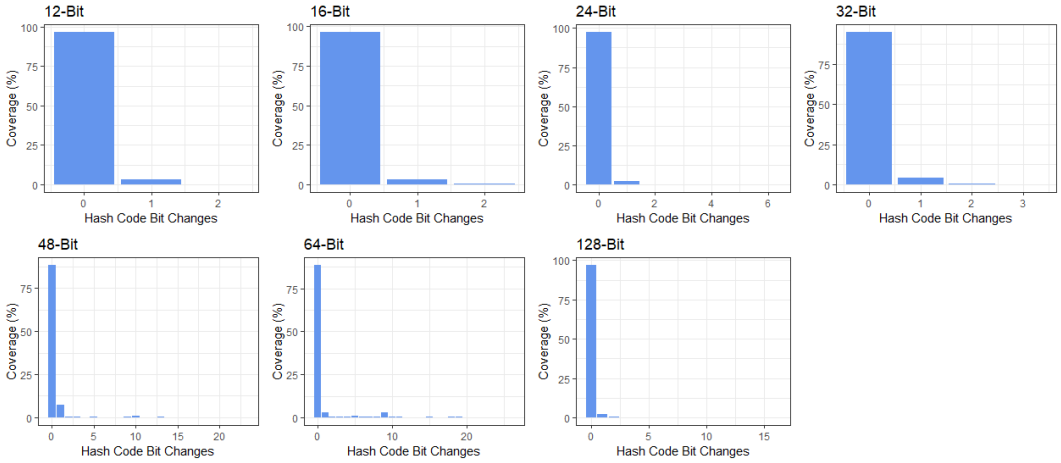


Figure 5.19: Coverage Variation for Neighboring Matches Amazon-Google (Single Feature Hashing) for different Code Lengths

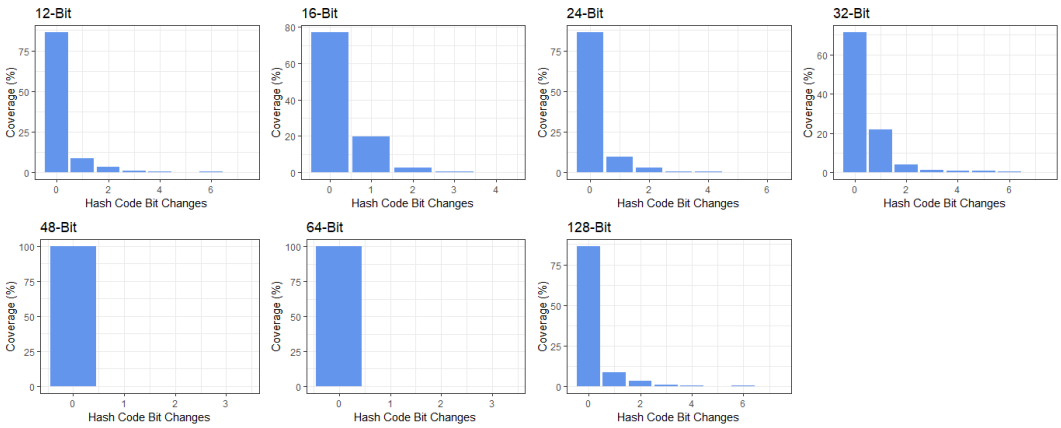


Figure 5.20: Coverage Variation for Neighboring Matches Amazon-Google (All Features Hashing) for different Code Lengths

– **DBLP-ACM**

Figure 5.21 depicts this plot for the DBLP-ACM dataset on Single Feature hashing and Figure 5.22 on All Features hashing. As seen in Figure 5.21, the two properties are satisfied by the code lengths 12, 16, 24, 32, 128 requiring up to 3-bit changes. Code Lengths 48 and 64, however, do not show a constant downward trend and spans up to 12+ bit changes. As seen in Figure 5.22, all the code lengths exhibit a downward trend and require no bit change with exact 100% coverage.

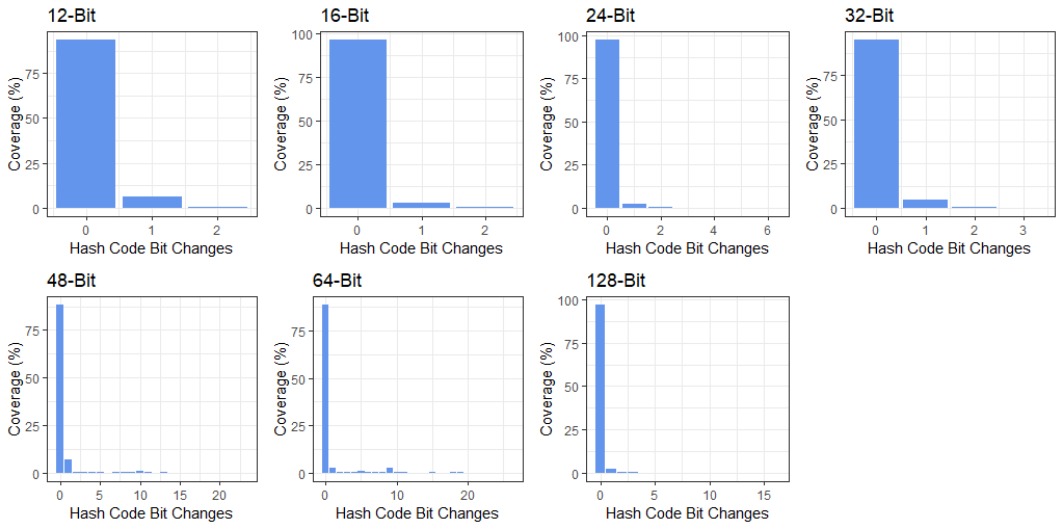


Figure 5.21: Coverage Variation for Neighboring Matches DBLP-ACM (Single Feature Hashing) for different Code Lengths

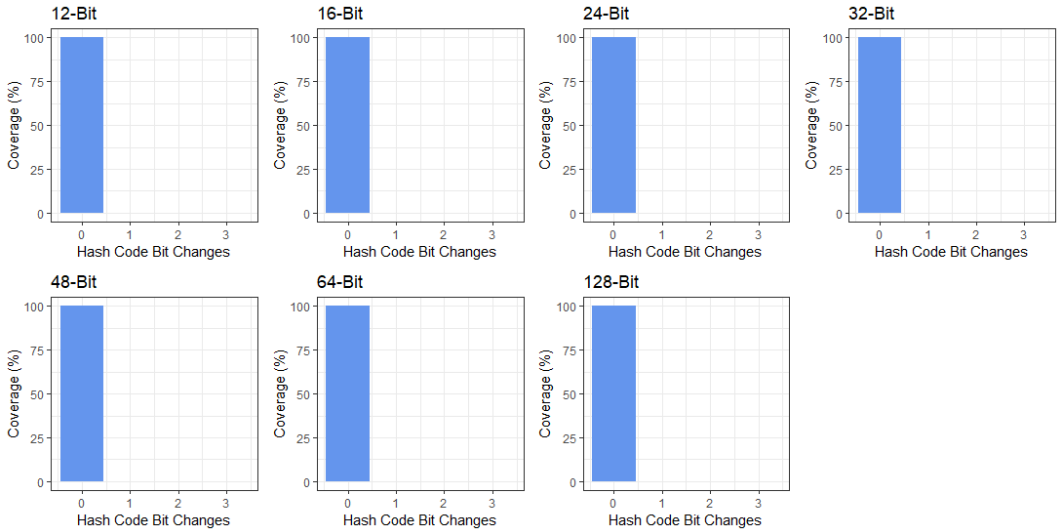


Figure 5.22: Coverage Variation for Neighboring Matches DBLP-ACM (All Features Hashing) for different Code Lengths

– **Walmart-Amazon**

Figure 5.23 depicts this plot for the Walmart-Amazon dataset on Single Feature hashing and Figure 5.24 on All Features hashing. As seen in Figure 5.23, the two properties are satisfied by the code lengths 12, 16, 24, 128 requiring up to 3-bit changes. Code Lengths 32, 48 and 64, however, do not show a constant downward trend and spans up to 9+ bit changes. As seen in Figure 5.24, the two properties are satisfied by the code lengths 12, 16, 24, 32, 48, 64 requiring up to 2+ bit changes. Code length 128, however, does not show a constant downward trend and spans up to 12-bit changes.

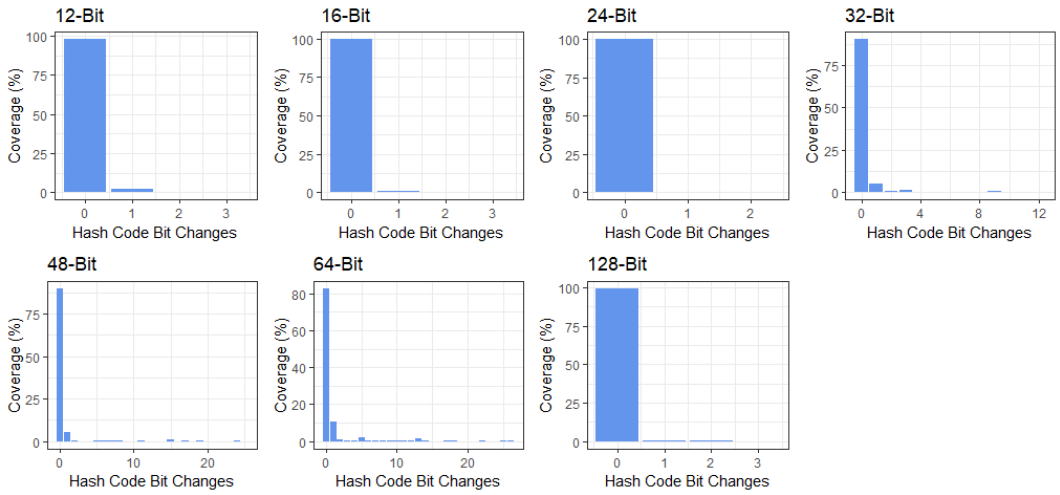


Figure 5.23: Coverage Variation for Neighboring Matches Walmart-Amazon (Single Feature Hashing) for different Code Lengths

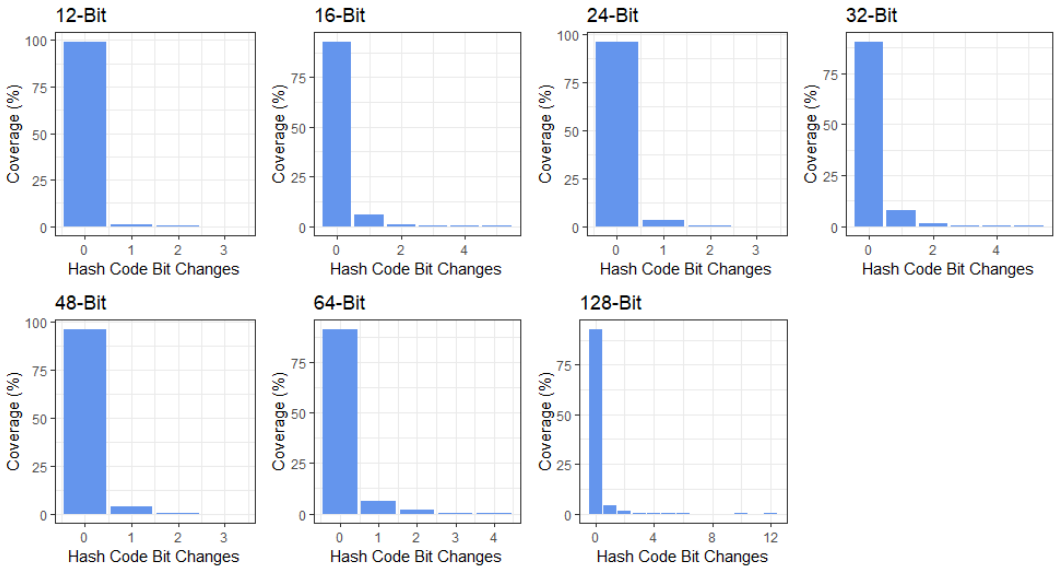


Figure 5.24: Coverage Variation for Neighboring Matches Walmart-Amazon (All Features Hashing) for different Code Lengths

- **Comparison with the Baseline Brute-Force Approach**

  Table 5.1 depicts the maximum coverage and minimum computation achieved through the brute-force approach in comparison to that achieved through L2H - Deep Hashing on all datasets for Single Feature and All Features hashing. We can observe that the brute-force approach always achieves the ideal coverage of 100% for single and all features hashing. In comparison L2H - Deep Hashing achieves considerable maximum coverage being 96%+, with few of them being 100% or almost 100%. The computation values between the two approaches do not show a significant difference and are comparable except for a significant difference observed between the computation value shown by DBLP-ACM All Features hashing.

- **Classification Accuracy for the Evaluation of the Task of Supervised Entity Resolution**

  Table 5.2 depicts the classification accuracy in the form of F1 score on all datasets for the Single Feature hashing at the *Top* and for All Features hashing at the *Bottom* for different Code Lengths. In doing so, we evaluate the performance of L2H - Deep Hashing for the task of supervised entity resolution as described in Chapter 2 Section 2.5.2.3. Further, the combinations marked in **bold** denote the best-performing ones from the set for the respective datasets. Comparing the classification performance of the datasets for Deep Hashing technique, DBLP-ACM dataset outperforms the two for both Single and All Features hashing as depicted in Figure 5.25. On the Single Feature hashing, we can report a result for this dataset that improves by several percentage points over the corresponding LSH evaluation; for the other case on this dataset, the result is similar to the LSH evaluation. For the other two datasets, blocking with L2H brings great improvements in the achieved F1 scores, with All Features hashing lagging behind

Parameter	Amazon-Google	DBLP-ACM	Walmart-Amazon
Brute-Force Approach Single Feature Hashing			
Coverage	100	100	100
Computation	0.029974	0.029974	0.029974
L2H - Deep Hashing Single Feature Hashing			
Coverage	97.461538	97.461538	99.792099
Computation	0.042937	0.042937	0.046967
Brute-Force Approach All Features Hashing			
Coverage	100	100	100
Computation	0.029974	0.037059	0.001732
L2H - Deep Hashing All Features Hashing			
Coverage	100	100	96.257796
Computation	0.048782	100	0.009173

Table 5.1: Comparison of Brute-Force Approach and L2H - Deep Hashing Hashing Technique with Single Feature Hashing and All Features Hashing

the Single Feature hashing for the Amazon-Google dataset, in the overall trend and in the best score achieved. For the Walmart-Amazon, it is more difficult to determine which configuration leads to the best overall F1 scores (between Single Feature or All Features hashing), however, Single Feature shows the precise best F1 score. Altogether, the impact of code length is not clear.

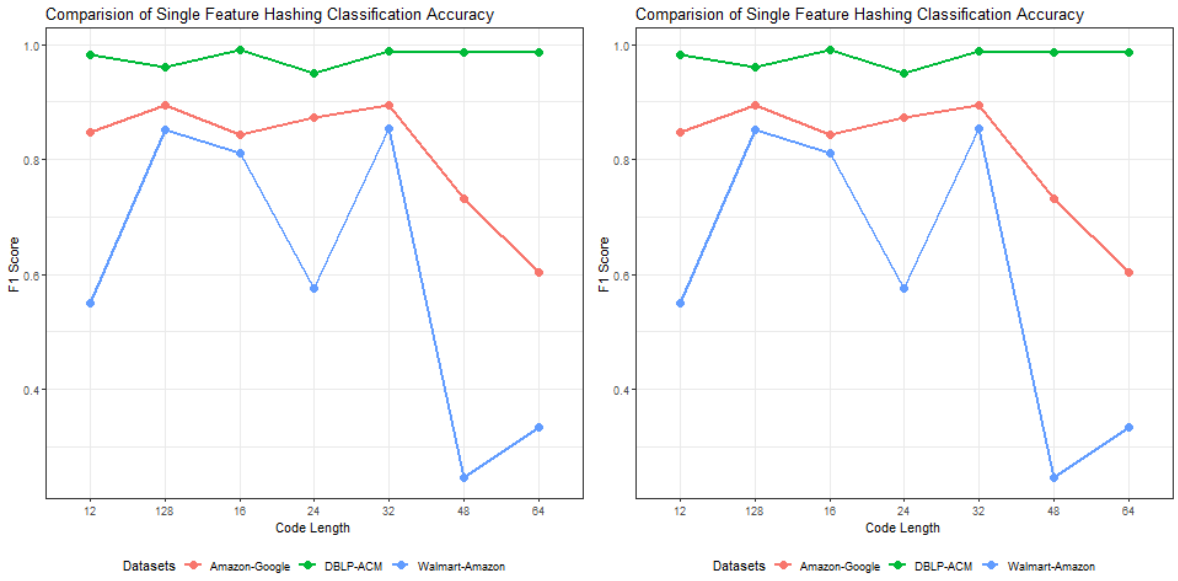


Figure 5.25: Comparison of Classification Accuracy on the Datasets

- **Comparison of LSH - Super-Bit and L2H - Deep Hashing**

  Table 5.3 reports the comparison between LSH - Super-Bit and L2H - Deep Hashing for Single Feature Hashing and All Features Hashing for different parameters, namely, highest coverage, minimum computation and highest F1 score reported during the experimental evaluation. Comparing these values, it can be observed that L2H - Deep Hashing clearly outperforms LSH - Super-Bit for a majority of the cases in terms of all the three values of Coverage, Computation and F1 Score. Specifically, it achieves desirable low computation values in comparison to that of LSH. It also significantly improvises on the low F1 scores (i.e., corresponding to the task of supervised entity resolution) obtained by LSH on Amazon-Google and Walmart-Amazon datasets and outputs F1 scores for all datasets with values of 0.8+. Considering the case of DBLP-ACM All Features hashing, the difference between the values for the two techniques where LSH - Super-Bit outperforms, it remains comparable. Same applies for the coverage values for the case of Walmart-Amazon all features hashing. For all datasets, the best F1 score was achieved for one configuration of L2H with Single Feature hashing.

Answering RQ$_2$, we experimentally evaluate L2H - Deep Hashing on the same parameters of Coverage, Computation and F1 Score as we evaluate LSH - Super-Bit in Chapter 4. Based

Code Length	Amazon-Google	DBLP-ACM	Walmart-Amazon
12	0.846584	0.982410	0.549565
16	0.842352	**0.990789**	0.811447
24	0.873619	0.949928	0.575396
32	**0.895281**	0.988126	**0.852807**
48	0.732163	0.987215	0.246648
64	0.602977	0.985487	0.334883
128	0.893662	0.960784	0.851351

Code Length	Amazon-Google	DBLP-ACM	Walmart-Amazon
12	0.285171	0.926128	0.470588
16	0.581267	0.928916	0.602189
24	0.687581	0.920612	0.667889
32	0.765799	0.927719	0.735593
48	0.298932	0.921885	0.787775
64	0.308219	0.931635	0.796084
128	**0.878363**	**0.936288**	**0.846280**

Table 5.2: F1 Scores for all Datasets reported for Single Feature hashing at the *Top* and for All Features hashing at the *Bottom*

Parameter	Amazon-Google	DBLP-ACM	Walmart-Amazon
LSH - Super-Bit Single Feature Hashing			
Coverage	87	94	94
Computation	66	50	26
F1 Score	0.333333	0.947791	0.242424
L2H - Deep Hashing Single Feature Hashing			
Coverage	**97.461538**	**97.461538**	**99.792099**
Computation	**0.042937**	**0.042937**	**0.046967**
F1 Score	**0.895281**	**0.990789**	**0.852807**
LSH - Super-Bit All Features Hashing			
Coverage	99	**100**	**100**
Computation	50	**85**	100
F1 Score	0.385297	**0.940267**	0.263440
L2H - Deep Hashing All Features Hashing			
Coverage	**100**	**100**	96.257796
Computation	**0.048782**	100	**0.009173**
F1 Score	**0.878363**	0.936288	**0.846280**

Table 5.3: Comparison of LSH - Super-Bit and L2H - Deep Hashing Hashing Techniques with Single Feature Hashing and All Features Hashing

on the experimental evaluation, we observe that L2H - Deep Hashing can outperform LSH - Super-Bit with high F1 scores (i.e, corresponding to the task of supervised entity resolution), maximum coverage and significantly low computation values observed on all the three dataset pairs, Amazon-Google, DBLP-ACM and Walmart-Amazon for a majority of the cases. Thus it can be inferred that the data-dependent hashing technique can perform better than that of our considered baseline of data-independent hashing technique LSH - Super-Bit.

5.4 Summary

Summarizing, similar to the LSH - Super-Bit evaluation, we demonstrate and experimentally evaluate L2H - Deep Hashing for word embeddings on three dataset pairs namely, Amazon-Google, DBLP-ACM and Walmart-Amazon for *Single Feature* and *All Features* hashing. In doing so, we utilize our designed neural network *Deep Hash Neural Net (DHNN)*, based on the literature and generalized to work with embedding data. We measure the performance of L2H - Deep Hashing through its evaluation on these datasets with three metrics including two user-defined metrics namely *Coverage* measuring the number of correct similar mappings to the same bucket by L2H - Deep Hashing based on the Ground Truth dataset acting as a benchmark for similarity & *Computation* capturing the block distribution by L2H - Deep Hashing and *F1 score* measuring the classification accuracy for similarity search using L2H - Deep Hashing. The evaluation compromised of seven different bit code lengths, including 12, 16, 24, 32, 48, 64, 128. Additionally, we depict the Coverage and Computation for each of the datasets through plots visualizing each of their relationship to the different hash code lengths. Furthermore, we consider the trade-off between Coverage and Computation for our experimental evaluation and visualize it through a Pareto Front plot depicting the Pareto Frontier optimal solutions for different hash code lengths.

We further utilize these hash code lengths to test for their classification accuracy based on the similarity measure *Cosine Similarity* between all the related features of the hashed vectors. In doing so, we evaluate the performance of L2H - Deep Hashing for the task of supervised entity resolution. Consequently, the performance of L2H - Deep Hashing can be tuned accordingly by adjusting the code length as suggested by the high classification accuracy, i.e., F1 score of a particular code length on the dataset. Furthermore observing upon the classification accuracy, similar to LSH - Super-Bit, DBLP-ACM dataset outputs the higher F1 score being almost close to 1 for all the selected Pareto Frontier solutions irrespective of Single Feature or All Features hashing. Furthermore, we also visualize the coverage variations for all the datasets for *Single Feature* as well as *All Features* hashing, to depict the bit changes required in the original hash code length to achieve the optimal coverage value. Additionally, we also compare the output Coverage and Computation values of L2H - Deep Hashing with our baseline of Brute-Force approach and observe comparable results for some of the cases.

Regarding performance comparison with LSH - Super-Bit, L2H - Deep Hashing shows a significant increase in the classification accuracy (i.e., corresponding to the task of supervised entity resolution) especially for the two dataset pairs Amazon-Google and Walmart-Amazon with F1 scores of 0.8+ as compared to those below 0.4 as exhibited by LSH. DBLP-ACM

dataset shows no significant difference in the F1 scores and for both techniques being in the range of 0.9+. L2H - Deep Hashing shows significantly reduced computation values, which serves as an added advantage over LSH - Super-Bit. Additionally, L2H - Deep Hashing also outputs high coverage values in comparison to LSH - Super-Bit for a majority of the cases. Additionally, L2H - Deep Hashing, as a data-dependent hashing technique, also eliminates the drawback of data-independent hashing techniques of requiring longer hash codes to achieve a reasonable amount of performance. In contrast, it can achieve a considerable performance with short similarity preserving hash codes as demonstrated by the L2H - Deep Hashing experimental evaluation. Our current results suggest that important future work should be considered, evaluating more thoroughly the impact of hyper-parameter tuning, the role of aspects that might impact the robustness of our solution, and establishing the effect of input size and code length on the performance of L2H.

To further effectively measure, compare and draw solid conclusions regarding the performance of L2H - Deep Hashing, we consider and evaluate how much speed-up this technique brings to real-time computations using top-k similarity search queries in Apache Spark in the next Chapter 6.

[1]https://cs.nju.edu.cn/lwj/slides/L2H.pdf

[2]https://xgboost.readthedocs.io/en/latest/index.html

6. Similarity Search Using Different File Formats in Apache Spark

In this chapter, we focus on the runtime-performance evaluation of Learning To Hash (L2H) - Deep Hashing in the context of top-k similarity search queries. Specifically, we evaluate and compare the execution-time benefits brought to top-k similarity search with hashed data using this technique to that without hashing in Apache Spark using different file formats.

This chapter is structured as follows:

- Section 6.1 recapitulates the research question of focus for this chapter.
- Section 6.2 presents an overview of our data pipeline to evaluate our L2H technique for similarity search using Apache Spark.
- Section 6.3 discusses and presents our evaluation results.
- Section 6.4 presents an effective summarization of the important aspects of this chapter.

6.1 Research Question

Recapitulating the research questions we established for this work, in this chapter, we will be addressing the following specific research question:

- **RQ**$_3$: What is the performance (execution time) speed-up achievable by hashing with the best-observed technique, for top-k searches in a large-scale processing framework? To what extent can partitioning by the corresponding hash codes affect the overall performance?

Relevance Measuring the similarity search query performance of the best-observed hashing technique, i.e., Learning To Hash (L2H) - Deep Hashing on a large scale processing framework, i.e., Apache Spark, helps to understand its real-time performance through the comparison of execution-time benefits in the considered context of search without hashing.

Description This research question deals with the experimental evaluation of the runtime-performance of Learning To Hash (L2H) - Deep Hashing in the context of top-k similarity search queries with its performance comparisons with the baseline search approach without data hashing (search without hash code). This specifically aims to evaluate how much execution time benefits are brought through searching based on hash code blocking versus that without hashing. In doing so, we also evaluate the performance improvements brought through the underlying data storage specifically as CSV, Parquet without Partition and Parquet with Partition (on hash code).

6.2 Data Pipeline

In our study on the evaluation of similarity search performance of Learning To Hash (L2H) - Deep Hashing, we primarily choose the data corresponding to the top two high F1 Score configurations for All Features hashing for each of the three dataset pairs in our L2H evaluation study in Chapter 5 (Refer Table 5.2). This results in the data corresponding to the hash code lengths 32-bit, 128-bit for Amazon-Google, 64-bit, 128-bit for DBLP-ACM and 64-bit, 128-bit for Walmart-Amazon which accumulates to form six input dataset pairs for our experimental study.

6.2.1 Data Storage

We highlighted on the different storage formats and its characteristics for large-scale processing in Chapter 2 Section 2.3.3.3. For our experimentation, we select to store the data on HDFS in three different file formats, the tabular textual layout as CSV (Row-based storage) and nested binary layout as Parquet (Columnar Storage) with and without partition. The Parquet files use the snappy compression technique, and the partitioned parquet files are partitioned based on hash code.

6.2.2 Evaluation

We want to experimentally evaluate the runtime query performance for top-k similarity search for deep hashing with different file formats in Apache Spark. In doing so, we consider for each of the layouts, benchmark evaluation on the baseline, i.e., top-k similarity search without hash code requiring a complete data scan. Specifically, we consider k = 10, and 20 different needles (items from one dataset, for searching on the other) separately for similarity search with hashing (without baseline) on each of the dataset in the pair. Further, we measure and record the average execution-time for the queries in seconds for the experimental comparisons with 20 repetitions per query.

Consequently, the evaluation criteria are as summarized below:

- **Baseline** Top-10 similarity search without hashing.
- **Evaluation Metrics** Average query execution time in seconds.

6.3 Results

In this section, we present and experimentally evaluate the results of our study on the six dataset pairs with different file formats in Apache Spark, thus answering our formulated RQ_3. As an illustration of the similarity search results, Figure 6.1 and Figure 6.2 depicts an example for the top-10 results retrieved through this query execution for baseline search without hash code and search with hash code for the DBLP-ACM parquet file format. The baseline execution typically requires a complete data scan affecting the query performance.

```
+------+--------------------+--------------------+------------------+
|    id|            hashcode|             overall|               sim|
+------+--------------------+--------------------+------------------+
|304576|01111101011111111...|-0.2025867,-0.045...|0.9999990806329542|
|253287|01111101011111111...|-0.07805355,0.123...|0.9999983195394684|
|375714|01111101011111111...|-0.13665967,0.062...|0.9999982849683622|
|191915|01111101011111111...|-0.015038077,-0.0...|0.9999982659655363|
|276318|01111101011111111...|-0.09740738,0.028...|0.9999982586379215|
|223896|01111101011111111...|-0.018436473,-0.0...|0.9999982409206319|
|304583|01111101011111111...|-0.14380206,0.136...|0.9999982374862542|
|253342|01111101011111111...|-0.051281296,0.03...|0.9999982370360729|
|276306|01111101011111111...|-0.053497355,-0.0...|0.9999982312482119|
|872846|01111101011111111...|-0.020269591,0.15...|0.9999982310223401|
+------+--------------------+--------------------+------------------+
```

Figure 6.1: Baseline (Search without Hash Code) Top-10 Similarity Search Results

```
+------+--------------------+--------------------+------------------+
|    id|            hashcode|             overall|               sim|
+------+--------------------+--------------------+------------------+
|304576|01111101011111111...|-0.2025867,-0.045...|0.9999990806329542|
|253287|01111101011111111...|-0.07805355,0.123...|0.9999983195394684|
|375714|01111101011111111...|-0.13665967,0.062...|0.9999982849683622|
|191915|01111101011111111...|-0.015038077,-0.0...|0.9999982659655363|
|276318|01111101011111111...|-0.09740738,0.028...|0.9999982586379215|
|223896|01111101011111111...|-0.018436473,-0.0...|0.9999982409206319|
|304583|01111101011111111...|-0.14380206,0.136...|0.9999982374862542|
|253342|01111101011111111...|-0.051281296,0.03...|0.9999982370360729|
|276306|01111101011111111...|-0.053497355,-0.0...|0.9999982312482119|
|872846|01111101011111111...|-0.020269591,0.15...|0.9999982310223401|
+------+--------------------+--------------------+------------------+
```

Figure 6.2: Hashing (Search with Hash Code) Top-10 Similarity Search Results

There are correspondingly three plots for each of the dataset pair corresponding to the three layouts, CSV, Parquet without Partition and Parquet with Partition. Accordingly, these are further depicted,

- **Amazon-Google Dataset** The evaluation results on Amazon-Google Dataset are as below:

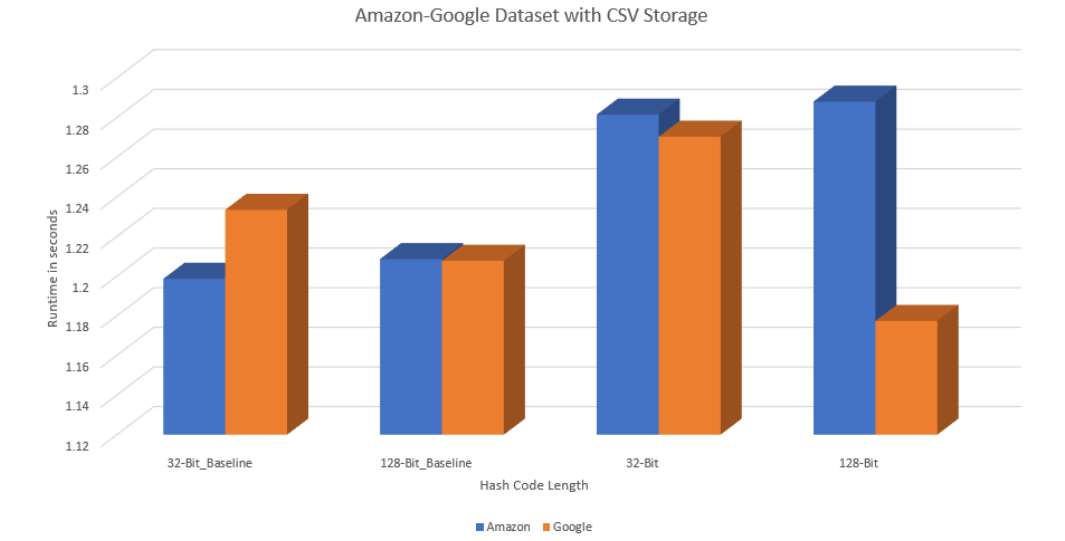


Figure 6.3: Execution Time for Amazon-Google Dataset with CSV Storage

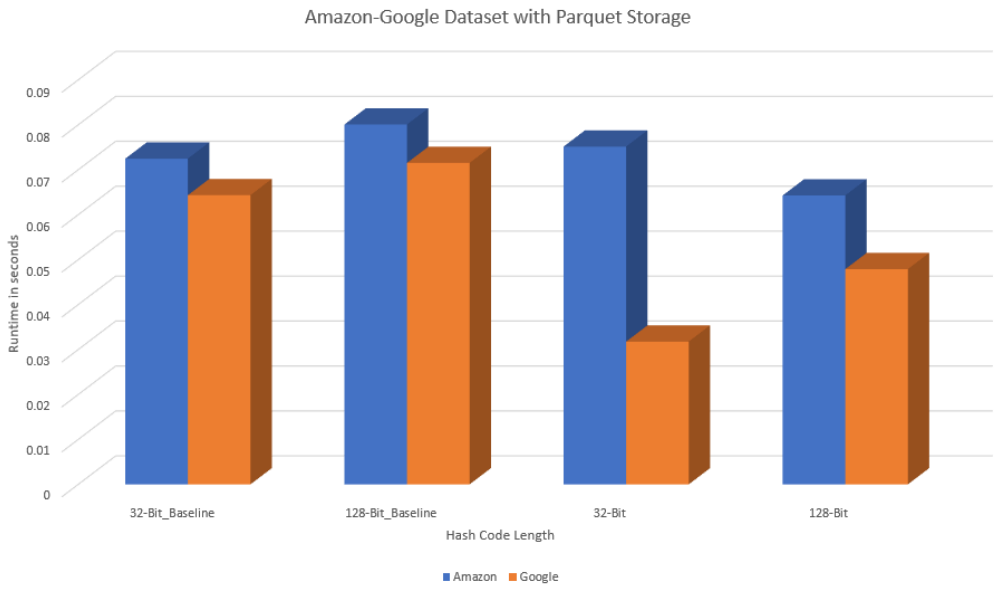


Figure 6.4: Execution Time for Amazon-Google Dataset with Parquet without Partition Storage

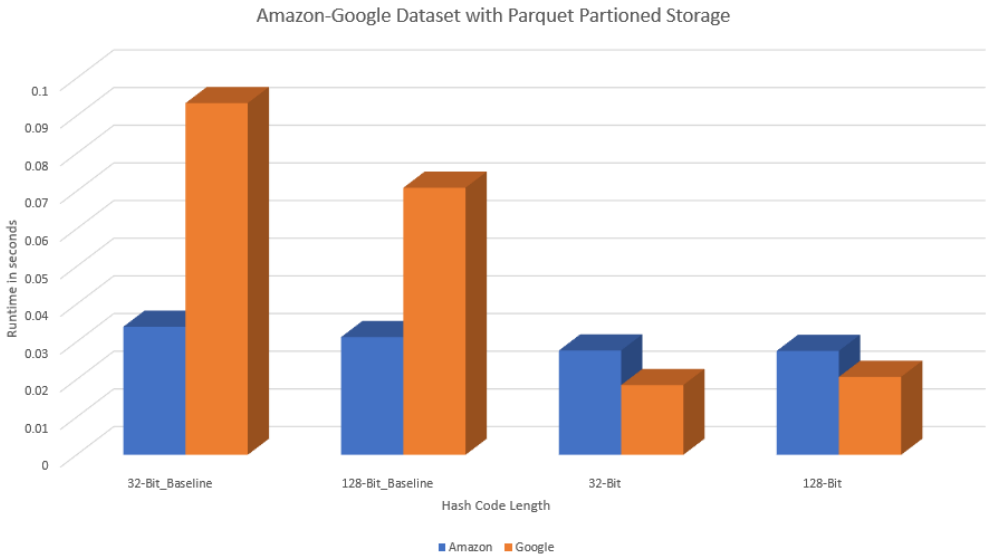


Figure 6.5: Execution Time for Amazon-Google Dataset with Parquet with Partition Storage

As observed in Figure 6.3, the performance benefits of hashing in comparison to that of non-hashing (baseline) is not depicted from the query execution time on CSV storage. However, if we look at Figure 6.4, the execution time with hashing is less than that of the baseline for parquet storage. Specifically, a hash code of 128-bit shows lower execution times for both the datasets in the pair. Further in the case of parquet partitioned storage, Figure 6.5 highlights on the efficient performance of hashing through the significantly reduced execution times as compared to the baseline for both the hash code lengths. If we compare the execution times with hashing across the considered file formats, its ranges from 1.17 - 1.28 sec for CSV, significantly reduced to 0.03 - 0.07 sec for parquet and reduced furthermore as 0.01 - 0.02 sec for parquet partition, highlighting the efficiency of parquet partitioned storage.

- **DBLP-ACM Dataset** The evaluation results on DBLP-ACM Dataset are as below:

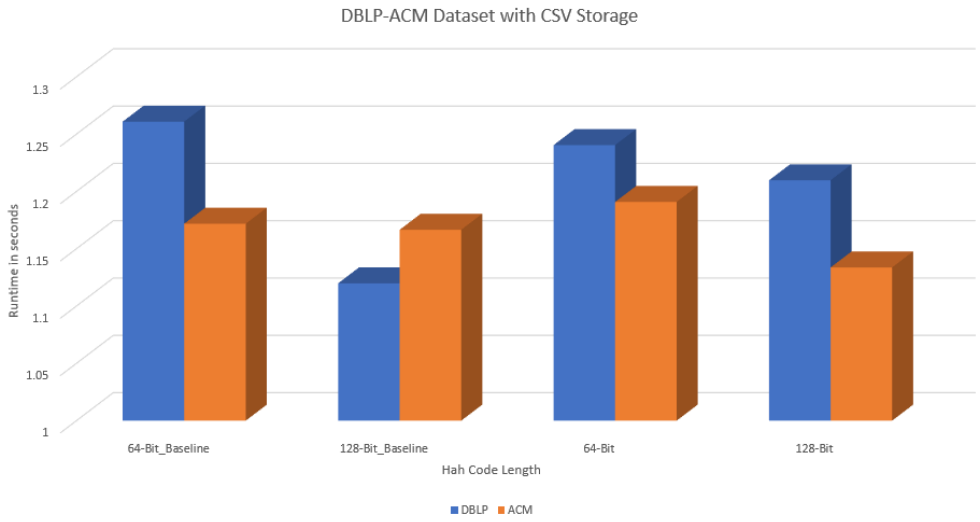


Figure 6.6: Execution Time for DBLP-ACM Dataset with CSV Storage

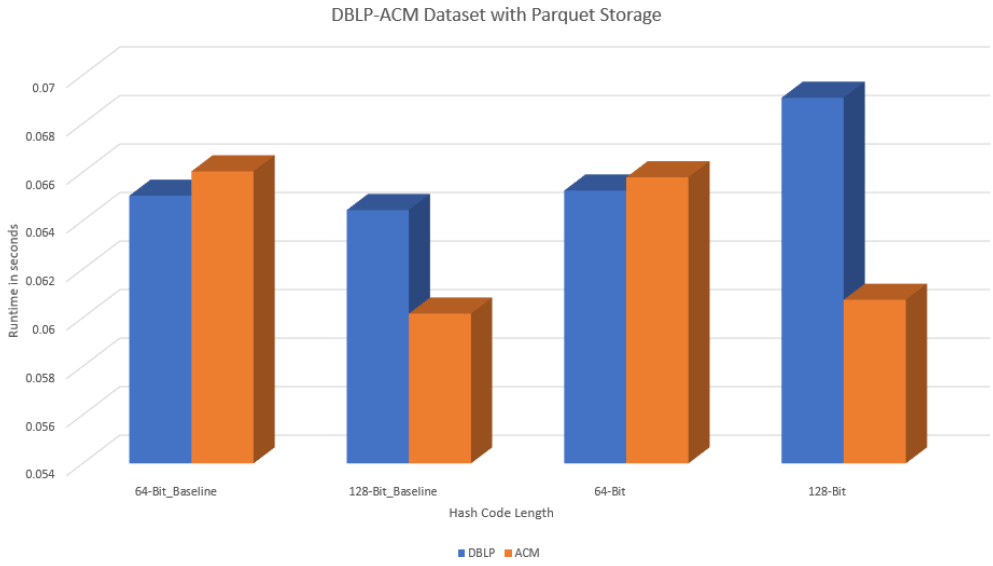


Figure 6.7: Execution Time for DBLP-ACM Dataset with Parquet without Partition Storage

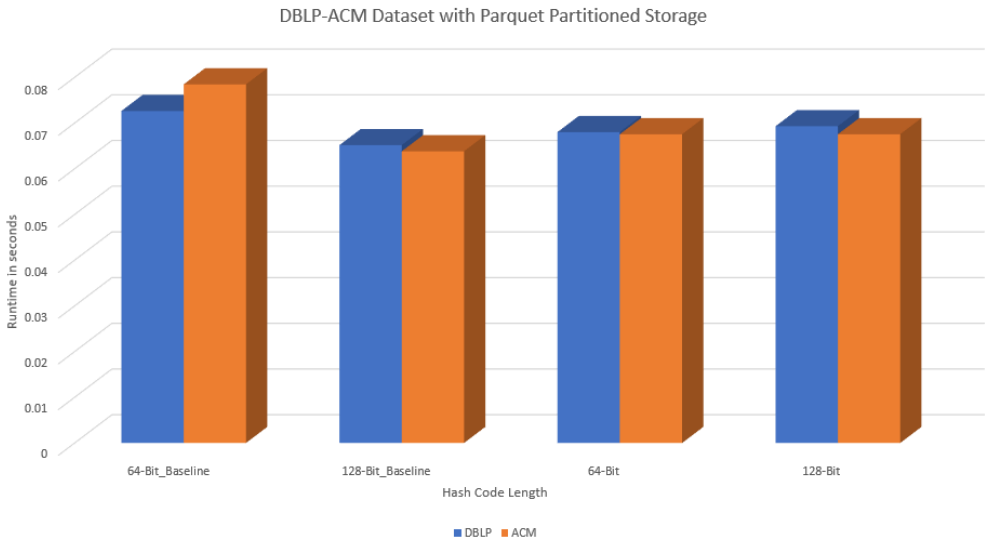


Figure 6.8: Execution Time for DBLP-ACM Dataset with Parquet with Partition Storage

As observed in Figure 6.6, the performance benefits of hashing in comparison to that of non-hashing (baseline) is partially depicted from the query execution time on CSV storage for 64-bit. The same case is reflected in Figure 6.7, where we have almost the same execution times with hashing and the baseline for parquet storage in the case of 64-bit. Further, in the case of parquet partitioned storage, Figure 6.8 depicts lower execution times for 64-bit and comparable in the case of 128-bit. If we compare the execution times with hashing across the considered file formats, its ranges from 1.13 - 1.24 sec for CSV, significantly reduced to the range of 0.06 - 0.07 sec for parquet with and without partition. The output results can be given the fact of the varied data distribution, DBLP-ACM in comparison to Amazon-Google and Walmart-Amazon contains highly reduced number of partitioning keys and also thus making no difference with the parquet with or without partition storage.

- **Walmart-Amazon Dataset** The evaluation results on Walmart-Amazon Dataset are as below:

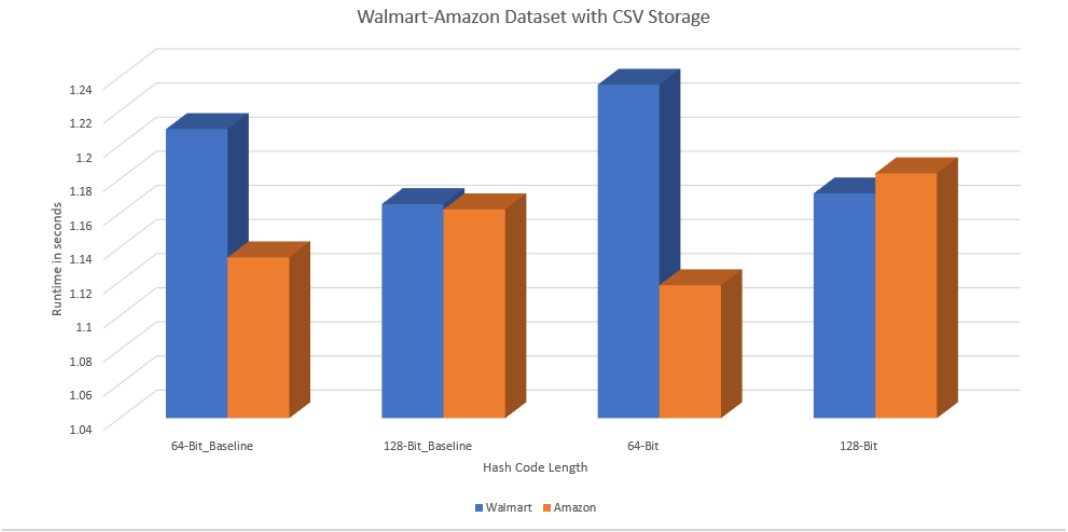


Figure 6.9: Execution Time for Walmart-Amazon Dataset with CSV Storage

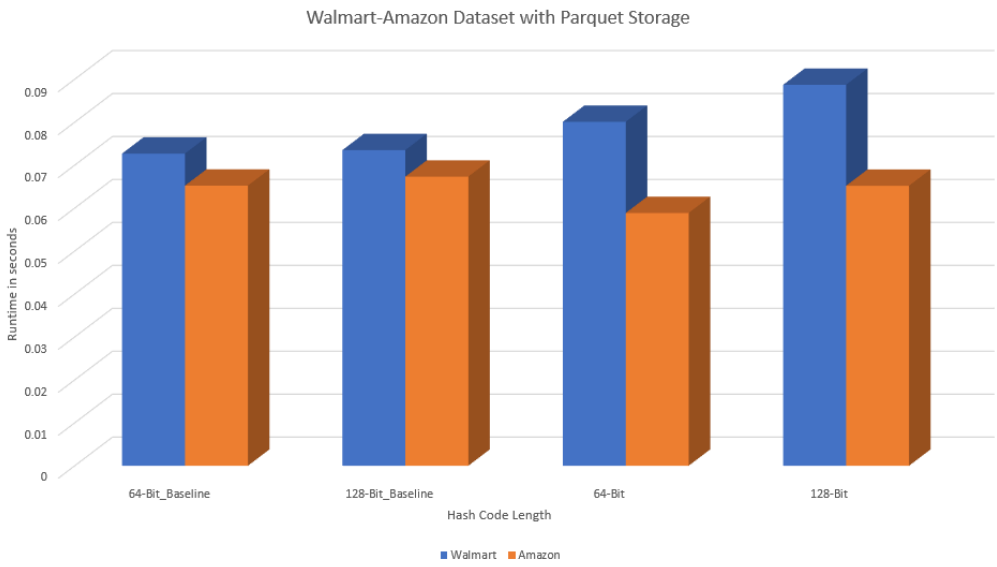


Figure 6.10: Execution Time for Walmart-Amazon Dataset with Parquet without Partition Storage

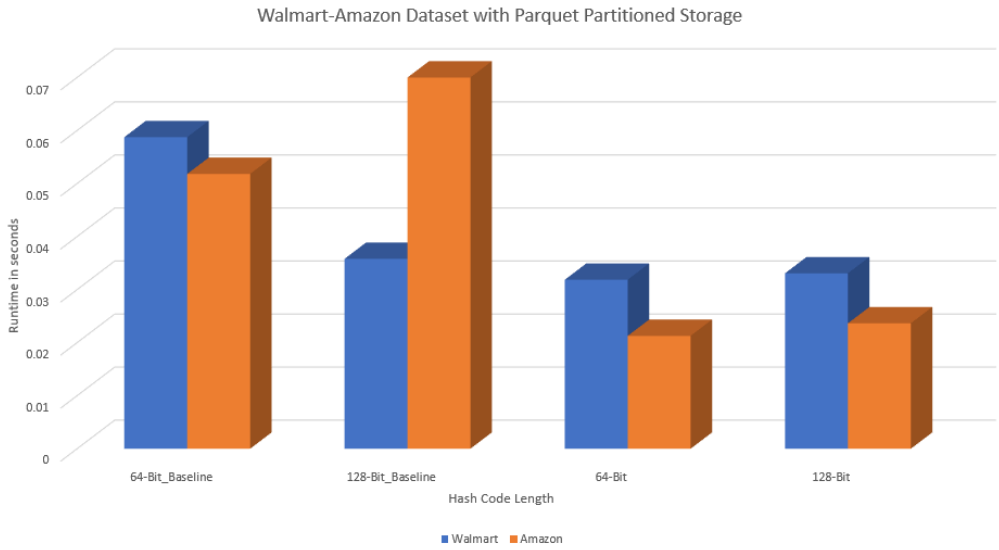


Figure 6.11: Execution Time for Walmart-Amazon Dataset with Parquet with Partition Storage

As observed in Figure 6.9, the performance benefits of hashing in comparison to that of non-hashing (baseline) is partially depicted from the query execution time on CSV storage for 64-bit. The same case is reflected in Figure 6.10, where we have almost the same execution times with hashing and the baseline for parquet storage in the case of 64-bit. However, in the case of parquet partitioned storage, Figure 6.11 highlights on the efficient performance of hashing through the significantly reduced execution times as compared to the baseline for both the hash code lengths. If we compare the execution times with hashing across the considered file formats, its ranges from 1.11 - 1.23 sec for CSV, significantly reduced to 0.05 - 0.08 sec for parquet and reduced furthermore as 0.02 - 0.03 for parquet partition, highlighting the efficiency of parquet partitioned storage.

- **Comparison of Runtime Performance for all Datasets** Figure 6.12 depicts the combined top-10 query execution time of each of the dataset in the pair. The depicted execution times are the least-recorded ones for the data layout for search based on the hash code. On observation of Figure 6.12, we see that for all the three dataset pairs, parquet with and without partition storage shows a notable reduction for the query execution time in comparison to that on CSV storage. Specifically, parquet with partition outputs the least query execution time depicting the increase in the search efficiency when the parquet file is partitioned based on the column to which the data is more likely to be queried against, i.e., hash code for our case. Consequently, this leads to high runtime-performance. On analysing further, we also find out that the distribution of the data also plays a major role in affecting the query execution time given the different file formats, i.e., parquet partition outputs reduced execution times in comparison to parquet without partition when we have a large number of partitioning keys as in case with Amazon-Google and Walmart-Amazon; else the performance is

comparable as in case of DBLP-ACM. Consequently comparing our datasets, from Figure 6.12, the order of efficient query execution time corresponding to high runtime performance for parquet partitioned can be determined as Amazon-Google, Walmart-Amazon and DBLP-ACM. Hence this time is correlated to the computation time metric we evaluated previously.

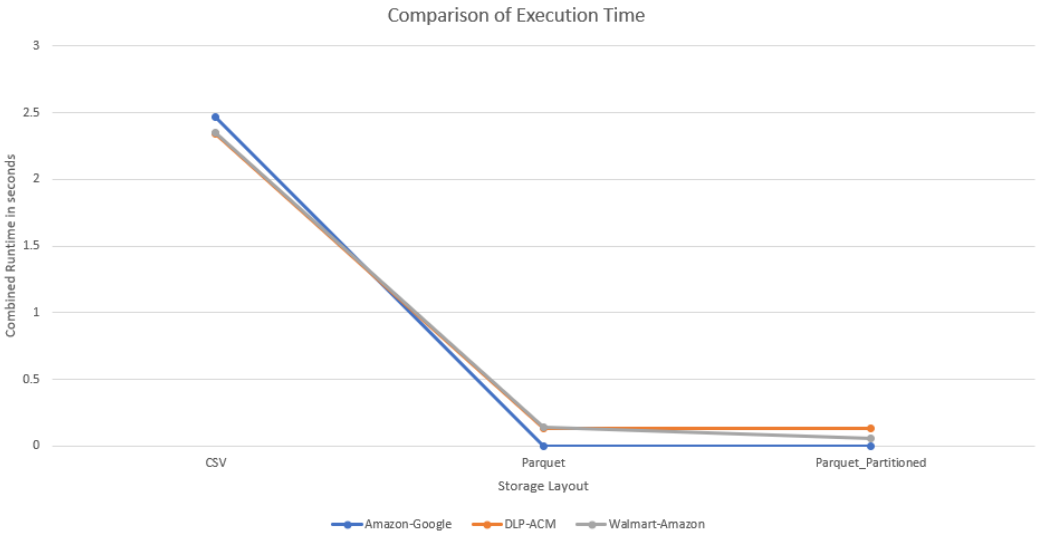


Figure 6.12: Comparison of Execution Time for all Datasets

Answering RQ$_3$, we experimentally evaluate L2H - Deep Hashing for its performance with top-10 similarity search queries as compared to the search without hashing approach (baseline) in Apache Spark with data stored in three different file formats on HDFS namely, CSV, Parquet, Parquet Partitioned. Based on the experimental evaluation, we observe that L2H - Deep Hashing reports significantly reduced execution times with the parquet storage in comparison to that of CSV. Specifically, L2H - Deep Hashing outputs significantly reduced execution times as compared to the baseline for both the considered high-performing (in terms F1 score) hash code lengths in the parquet partitioned layout. Thus it can be inferred that, L2H - Deep Hashing does significantly reduce the search time (as observed in the case of Amazon-Google and Walmart Amazon datasets) in the context of top-k similarity search as compared to the search without hashing when we have an underlying optimal data layout i.e., in our case, parquet partitioned through partitioning the data based on the search key (hash code). Additionally, in the same context, for DBLP-ACM dataset, this runtime was comparable to the baseline as DBLP-ACM contains very less number of partitioning keys when compared to that of Amazon-Google or Walmart-Amazon. However, this highlights the efficiency of L2H - Deep Hashing with parquet partition in case of real-time application data search and processing where we usually have high-dimensional varied data resulting in a high number of partitioning keys. Additionally, for this task, using the large-scale distributed cluster computing framework Apache Spark, allows for fast and without overhead query processing on Hadoop data through Spark SQL and additional performance benefits as highlighted in Chapter 2.

6.4 Summary

Summarizing, through our experimental evaluation, we demonstrate the real-time performance improvements brought by Learning To Hash - Deep Hashing on similarity search queries using Apache Spark. Specifically, we consider the hashed data corresponding to the top two F1 scores for All Features Hashing on the datasets. This resulted in the data corresponding to the hash code lengths 32-bit, 128-bit for Amazon-Google, 64-bit, 128-bit for DBLP-ACM and 64-bit, 128-bit for Walmart-Amazon. We further consider 20 top-10 similarity search queries on each of the datasets in the pair and evaluated its average runtime against that of search without hashing as a baseline. Additionally, we also consider data stored in three different file formats on HDFS to access their impact on query performance.

Consequently, we observe that in comparison to CSV storage, parquet with and without partitioned storage outputs significantly reduced runtime offering better query performance. Specifically, for the parquet partitioned storage, datasets Amazon-Google and Walmart-Amazon demonstrate significantly reduced runtimes in comparison to the baseline for both the hash code lengths and comparable runtimes were reported for the DBLP-ACM dataset. This highlights the efficiency of the deep hashing technique with the parquet partitioned storage. This infers the fact that parquet files offer data skipping thus scanning fewer relations in the case of conditional query execution and provide for runtime performance benefits especially when the data is partitioned based on the search key, i.e., in our case with the hash code. Thus as we demonstrate on the efficiency of L2H - Deep Hashing in comparison to LSH - Super-Bit resulting in the generation of effective hash codes in Chapter 5, this experiment further evaluates on the performance of Deep Hashing, highlighting its effectiveness with the parquet partitioned storage with significant reduced execution times as compared to the baseline without hashing search.

7. Related Work

In this chapter, we consider related work in light of the research work carried out in our thesis.

We structure this chapter into three brief sections, as follows:

- We begin by considering work in the area of blocking techniques for entity resolution, in Section 7.1.
- Subsequently we consider work that studies supervised deep hashing, in Section 7.2.
- We conclude the chapter by considering work on high-dimensional similarity search, in Section 7.3.

7.1 Blocking Techniques for Entity Resolution

Blocking methods partition the data into blocks mainly based on the similarity attributes, making it feasible for data skipping leading to a less number of record comparisons for the task of entity resolution. Relevantly, Baxter et al. [230] highlight, compare and evaluate the different types of blocking methods namely, *Bigram Indexing*, *Canopy Clustering with TF-IDF*, *Standard Traditional Blocking*, and *Sorted Neighbourhood Blocking* in the context of record linkage systems. Further, Knopke et al. [227] are among the first authors to carry out entity resolution with the use of classifiers. Their approach is based on the blocking of keys with experimental evaluation on two domains, namely, Bibliographic and E-commerce. The authors of this paper are also the creators of the datasets DBLP-ACM and Amazon-Google, which we used in our thesis research experiments. Saeedi et al. [231] give an example of another research work highlighting the approach of blocking of keys and the use of classifiers mainly in the form of scalable clustering for entity resolution. Further, the experimental evaluation is carried out on three domains, namely, Geographical, Music,

and Persons. Ebraheem et al. [232] propose a deep learning based approach to entity resolution and is the one closely related to our thesis work. This work also utilizes the datasets that we used in our thesis, but with varying attributes and with a different approach to embedding these attributes. Moreover, the proposed approach also utilizes a Locality Sensitive Hashing technique for blocking. Relevantly, the majority of the tasks covered by this work are similar to our thesis research work, though there is no study of the impact of locality-sensitive hashing configurations. Mugdal et al. [233] author a state-of-the-art paper for entity resolution using deep learning methods, however, there is little focus given to the blocking approach. More recently, Chen et al. [234] propose and experimentally evaluate a hybrid approach for entity resolution using embeddings (from the pre-trained fastText models as well) and traditional hand-curated similarity measures. This research also proposes to consider blocking in this hybrid approach as a future direction. In sum, research in blocking for entity resolution has not studied in depth the impact of locality sensitive hashing. Furthermore, though there is progress in the use of deep neural networks for the supervised entity resolution task ([232][233]), or for embedding data ([234]), there is no consideration to date on using deep neural networks to support the blocking process, as like we evaluate in our work.

7.2 Supervised Deep Hashing

In the context of supervised deep hashing methods, Li et al. [9] propose a novel Deep Pairwise-Supervised Hashing (DPSH), utilizing pairwise labels to simultaneously perform feature and hash code learning in an end-to-end architecture based on a feedback mechanism to learn better hash codes. Subsequently, Wang et al. [10] propose a novel triplet based deep hashing method, utilizing triplet labels to simultaneously performs image feature and hash code learning in an end-to-end manner, aiming to maximize the likelihood of the input triplet labels. Consequently, the design of our Deep Hash Neural Net (DHNN) utilizing triplet labels is influenced directly by this work [10]. The mentioned related works are designed for large-scale image retrieval tasks; however, we have designed a generalized deep hashing neural network to work with embedding data. As a result, our evaluation is also different, and we consider the execution time aspects of data skipping.

7.3 Efficient Techniques for High-Dimensional Similarity Search

Faiss[1][235] is a library from Facebook, Inc.[2]that supports high-dimensional similarity search. The library consists of algorithms which utilize GPU's for similarity search that achieve near-optimal performance. Additionally, SciDB [27, 29] as a column-oriented database management system is designed to support and analyse multi-dimensional arrays with any number of dimensions. To our knowledge, the SciDB research has not proposed any specialized approach for similarity-sensitive data distribution, nor has the Faiss library been used to study the use of neural networks for similarity-preserving hashing.

[1]https://github.com/facebookresearch/faiss

[2]https://research.fb.com/category/facebook-ai-research/

8. Conclusion and Future Directions

In this chapter, we present the core conclusions derived from this thesis work. We also propose some future directions for this thesis research.

This chapter is structured as follows:

- Section 8.1 presents the key conclusions derived from our thesis work.
- Section 8.2 discusses possible future work in our thesis area of research.

8.1 Conclusion

The research of this thesis seeks to contribute towards the efficient management of large-scale dense high-dimensional data, supporting similarity search. Specifically, we aim to help improve the data analytics pipeline for the utilization and effective management of embedding data, by analyzing Learning To Hash (L2H) as a solution to overcome the challenges (e.g., high class imbalance and the need for data skipping) posed by Big Data for Machine Learning applications. Accordingly, we design and implement a data-dependent Deep Hashing solution, *Deep Hash Neural Net (DHNN)* influenced from the literature. We evaluate comparatively our solution alongside a data-independent Locality Sensitive Hashing (LSH) alternative, Super-Bit, for the tasks of accelerating high-dimensional top-k similarity search on embedded data, and for the blocking step of supervised entity resolution. We employ three entity resolution dataset pairs, from two different domains, namely, Bibliographic and E-commerce with varying attributes, including, Amazon-Google, DBLP-ACM, and Walmart-Amazon, and embedded using fastText neural model. We evaluate the two hashing techniques for their performance based on two user-defined metrics, Coverage and Computation, and further measure its classification performance using F1 score (i.e., corresponding to the task of supervised entity resolution).

We report that L2H is able to help in the data distribution and achieve higher F1 scores than the LSH baseline, for a fixed competitive supervised learning process. We further conclude our experimental evaluation study of Learning To Hash - Deep Hashing with the measurement of its runtime performance in Apache Spark, validating its potential for speeding up the computation for top-k similarity search queries using different file formats.

8.2 Future Directions

Based on the literature, research involving efficient techniques for large-scale high-dimensional similarity search can be considered as a prominent area of research, and hence this leaves our thesis with a lot of scope for future work.

Neural network aspects: Future directions for this thesis work include, experimentation and evaluation of the proposed deep hashing method utilizing Deep Hash Neural Net (DHNN) with datasets from different domains with varying attributes. In this task, the adaptation of the network architecture and hyper-parameters to the dataset requires detailed study. Reports on the robustness, the ability to replicate the exact same results from different training sessions with the same architecture, are important. In our work, given the variety of datasets and training configurations, we trained the network once per case. Relevant future work can include to train this network several times to test the robustness of the learning process. Additionally, further improvisation through experimentation to fine-tune the loss function for our study also constitutes a relevant future work.

Similarity-search aspects: Detailed accuracy comparisons of deep hashing with the considered baseline of without data hashing similarity search can be included as future work. In addition, comparing against LSH is also an important direction. Furthermore, measurement of changes in the runtime performance of deep hashing through the experimentation with Apache Spark different performance tuning parameters can also be a future direction in this research. Finally, a comparative evaluation of the deep hashing method with Faiss[1][235] with regards to time required for the high-dimensional similarity search tasks can be considered in the future as a baseline experimental evaluation.

Supervised entity resolution aspects: Follow-up work should evaluate the entity resolution task while using more datasets, a different assortment of classifiers and train/test configurations. Additionally, a network-oriented perspective on the data, including network-related features, could be studied, perhaps extending our research to applications in graph-processing frameworks or using graph embeddings.

Large-scale processing aspects: In our work, we studied the different hashing methods for improving applications in a dataflow engine. Future work might also bring insights into the benefits of these methods in alternative processing frameworks, such as streaming engines. The adoption of hybrid engines like Weld or Musketeer could contribute to a common implementation being evaluated over diverse frameworks. Additionally, the integration with storage engine internals seems like a potentially worthwhile area of research, since it would

leverage better the similarity-preserving hashing, helping researchers in exploring how to leverage automated data skipping for this kind of data and application.

[1]https://github.com/facebookresearch/faiss

9. Appendix 1: Derivation of Loss Function

In this section we present the step by step derivation of the loss function used in our model, as presented in Equation 3.3, in Chapter 3 Section 3.3.2.

We start, by recollecting the definition of the target likelihood of pairwise labels $\mathcal{S} = \{s_{ij}\}$:

$$p(s_{ij}|\mathcal{B}) = \begin{cases} \sigma(\Theta_{ij}), s_{ij} = 1 \\ 1 - \sigma(\Theta_{ij}), s_{ij} = 0 \end{cases} \tag{9.1}$$

This definition encapsulates the goal of having highly similar hash codes (where the inner product is high), with a probability tending to 1, and highly different hash codes, with a probability tending to 0.

The following equation defines a loss function, such that the learnable parameters (the neural network, in charge of assigning the hash codes) can be tuned. The intention of this loss function is to help maximize the likelihood of similar items having a high probability, if applying logarithms to it, this is the log-likelihood. Maximizing this likelihood is the same as minimizing the negative loss likelihood, which is the core of the following equation:

$$\mathcal{L} = -\log p(\mathcal{S}|\mathcal{B}) \tag{9.2}$$

Replacing Equation 9.2 by how it is applied for each item in a batch:

$$\mathcal{L} = -\sum_{s_{ij} \in \mathcal{S}} \log p(s_{ij}|\mathcal{B}) \tag{9.3}$$

By substituting each item for the sigmoid function definition ($\sigma(\Theta_{ij}) = \frac{1}{1+e^{-\Theta_{ij}}}$), we obtain the following single equation, which captures the different ways of calculating the probability, according to the similarities (as defined in Equation 9.1):

$$\mathcal{L} = -\sum_{s_{ij} \in \mathcal{S}} \log((\frac{1}{1+e^{-\Theta_{ij}}})^{s}(1-\frac{1}{1+e^{-\Theta_{ij}}})^{1-s}) \tag{9.4}$$

Applying log arithmetic rules, to change the exponentiation into a product, and a product into a sum, we obtain:

$$\mathcal{L} = -\sum_{s_{ij} \in \mathcal{S}} s\log(\frac{1}{1+e^{-\Theta_{ij}}}) + (1-s)\log(1-\frac{1}{1+e^{-\Theta_{ij}}}) \tag{9.5}$$

With further arithmetic changes, to transform the divisions into subtractions, and aggregating the rightmost item into a fraction, we obtain:

$$\mathcal{L} = -\sum_{s_{ij} \in \mathcal{S}} s\log(1) - s\log(1+e^{-\Theta_{ij}}) + (1-s)\log(\frac{e^{-\Theta_{ij}}}{1+e^{-\Theta_{ij}}}) \tag{9.6}$$

Canceling out the leftmost item (as it evaluates to 0), and solving the multiplication in the rightmost item:

$$\mathcal{L} = -\sum_{s_{ij} \in \mathcal{S}} -s\log(1+e^{-\Theta_{ij}}) + \log(\frac{e^{-\Theta_{ij}}}{1+e^{-\Theta_{ij}}}) - s\log(\frac{e^{-\Theta_{ij}}}{1+e^{-\Theta_{ij}}}) \tag{9.7}$$

Expanding the rightmost item with the arithmetic rule for logs of divisions:

$$\mathcal{L} = -\sum_{s_{ij} \in \mathcal{S}} -s\log(1+e^{-\Theta_{ij}}) + \log(\frac{e^{-\Theta_{ij}}}{1+e^{-\Theta_{ij}}}) - s\log(e^{-\Theta_{ij}}) + s\log(1+e^{-\Theta_{ij}}) \tag{9.8}$$

Canceling out the leftmost and rightmost elements, we obtain:

$$\mathcal{L} = -\sum_{s_{ij} \in \mathcal{S}} \log(\frac{e^{-\Theta_{ij}}}{1+e^{-\Theta_{ij}}}) - s\log(e^{-\Theta_{ij}}) = -\sum_{s_{ij} \in \mathcal{S}} \log(\frac{e^{-\Theta_{ij}}}{1+e^{-\Theta_{ij}}}) + s\Theta_{ij} \tag{9.9}$$

Considering that the leftmost item is an alternative expression for the sigmoid function ($\sigma(\Theta_{ij}) = \frac{1}{1+e^{-\Theta_{ij}}} = \frac{e^{\Theta_{ij}}}{1+e^{\Theta_{ij}}}$), we can replace such expression, as follows:

$$\mathcal{L} = -\sum_{s_{ij} \in \mathcal{S}} \log(\frac{1}{1+e^{\Theta_{ij}}}) + s\Theta_{ij} \tag{9.10}$$

Finally, by applying arithmetic rules for logarithms, and removing items that evaluate to zero, we obtain the following equation, which is the same as Equation 3.3:

$$\mathcal{L} = -\sum_{s_{ij} \in \mathcal{S}} s\Theta_{ij} - \log(1 + e^{\Theta_{ij}}) \tag{9.11}$$

10. Appendix 2: Prototypical Implementation of a Deep Hash Network

Code Listing 10.1: Definition of our Deep Hash Network

```
import tensorflow as tf
slim = tf.contrib.slim

def deep_hash_network(code_length, network_type, input):
  net = slim.fully_connected(input, 5120, activation_fn=tf.nn.relu)
  hash_code = slim.fully_connected(net, code_length, activation_fn=None)
  return network_type(hash_code)
```

Code Listing 10.2: Definition of the forward pass function

```
def _network_template(state):
  return our_net(code_length, collections.namedtuple('DQH_network', ['hash_values']),
                                                state)
batch_outputs1=[]
batch_outputs2=[]
batch_outputs3=[]

def _build_network():
  global batch_outputs1, batch_outputs2, batch_outputs3
  net= tf.make_template('network', _network_template)
  """
  Note that instead of applying sign to the values of the output,
  we clip here by -1 and 1.
  """
  batch_outputs1=tf.clip_by_value(net(states1_ph),-1.,1.)
  batch_outputs2=tf.clip_by_value(net(states2_ph),-1.,1.)
  batch_outputs3=tf.clip_by_value(net(states3_ph),-1.,1.)
```

Code Listing 10.3: Definition of the training function, following our design for the loss calculation

```
optimizer=tf.train.RMSPropOptimizer(learning_rate=0.0001)

"""
This defines our training operation, based on: Li, Wu-Jun, Sheng Wang, and Wang-
Cheng Kang. "Feature learning based deep supervised hashing with pairwise labels."
arXiv preprint arXiv:1511.03855 (2015).
However we extend it to a triple, because we observed it empirically to accelerate
training.
"""

def _build_train_op():
 theta=tf.divide(tf.reduce_sum(tf.multiply(batch_outputs1[0],batch_outputs2[0]),1),2)
 theta2=tf.divide(tf.reduce_sum(tf.multiply(batch_outputs1[0],batch_outputs3[0]),1),2)
 theta3=tf.divide(tf.reduce_sum(tf.multiply(batch_outputs2[0],batch_outputs3[0]),1),2)

 sim_loss=-tf.reduce_sum(-tf.math.log(1+tf.math.exp(theta))+3*theta , 0)
 disim_loss1=-tf.reduce_sum(-tf.math.log(1+tf.math.exp(theta2)), 0)
 disim_loss2=-tf.reduce_sum(-tf.math.log(1+tf.math.exp(theta3)), 0)
 loss=sim_loss+disim_loss1+disim_loss2

 with tf.control_dependencies([]):
   gvs = optimizer.compute_gradients(loss)
   capped_gvs = [(tf.clip_by_value(grad, -10., 10.), var) for grad, var in gvs]
   #We clip by value to avoid exploding gradients
   return optimizer.apply_gradients(capped_gvs)
```

Code Listing 10.4: Example use for training

```
with tf.device(tf_device):
  batch_outputs1=tf.placeholder(tf.float32, name='bo1_ph')
  batch_outputs2=tf.placeholder(tf.float32, name='bo2_ph')
  batch_outputs3=tf.placeholder(tf.float32, name='bo3_ph')
  states1_ph = tf.placeholder(tf.float32, (None,size_of_embedding), name='state1_ph')
  states2_ph = tf.placeholder(tf.float32, (None,size_of_embedding), name='state2_ph')
  states3_ph = tf.placeholder(tf.float32, (None,size_of_embedding), name='state3_ph')
  net= _build_network()
  _train_op = _build_train_op()

for epoch in range(0,num_epochs):
  [result]=_sess.run([_train_op], feed_dict={states1_ph: np.array(amazon_p1,dtype=np.
                                             float64), states2_ph: np.array(google_p2,
                                             dtype=np.float64), states3_ph: np.array(
                                             disim_n,dtype=np.float64)})
```

Code Listing 10.5: Example use for generating hash codes

```
batch= np.sign(_sess.run(batch_outputs1, {states1_ph: np.array(amazon_keys,dtype=np.
                                          float64), states2_ph:None, states3_ph: None
                                          })[0])
```

Bibliography

[1] A. L'heureux, K. Grolinger, H. F. Elyamany, and M. A. Capretz, "Machine learning with big data: Challenges and approaches," *IEEE Access*, vol. 5, pp. 7776–7797, 2017. (cited on Page 5, 1, 3, and 10)

[2] N. W. Grady, "Kdd meets big data," in *2016 IEEE International Conference on Big Data (Big Data)*, pp. 1603–1608, IEEE, 2016. (cited on Page 5 and 4)

[3] S. Pal, *SQL on Big Data: Technology, Architecture, and Innovation*. Apress, 2016. (cited on Page 5, 10, 16, and 18)

[4] C. Doulkeridis and K. NØrvåg, "A survey of large-scale analytical query processing in mapreduce," *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 23, no. 3, pp. 355–380, 2014. (cited on Page 5, 10, and 20)

[5] S. Salloum, R. Dautov, X. Chen, P. X. Peng, and J. Z. Huang, "Big data analytics on apache spark," *International Journal of Data Science and Analytics*, vol. 1, no. 3-4, pp. 145–164, 2016. (cited on Page 5, 10, 20, and 21)

[6] M. Hausenblas, "Notes on physical & logical data layouts," *arXiv preprint arXiv:1305.6506*, 2013. (cited on Page 5, 10, 28, and 29)

[7] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, *et al.*, "Mllib: Machine learning in apache spark," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1235–1241, 2016. (cited on Page 5, 10, 20, 35, and 36)

[8] J. Ji, J. Li, S. Yan, B. Zhang, and Q. Tian, "Super-bit locality-sensitive hashing," in *Advances in Neural Information Processing Systems*, pp. 108–116, 2012. (cited on Page 5, 10, 39, 40, 52, and 64)

[9] W.-J. Li, S. Wang, and W.-C. Kang, "Feature learning based deep supervised hashing with pairwise labels," *arXiv preprint arXiv:1511.03855*, 2015. (cited on Page 5, 10, 41, 42, 53, 54, and 126)

[10] X. Wang, Y. Shi, and K. M. Kitani, "Deep supervised hashing with triplet labels," in *Asian conference on computer vision*, pp. 70–84, Springer, 2016. (cited on Page 5, 10, 41, 42, 52, 53, 54, 90, and 126)

[11] H. Zhu, M. Long, J. Wang, and Y. Cao, "Deep hashing network for efficient similarity retrieval," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016. (cited on Page 5, 10, 41, and 42)

[12] X. Chen, E. Schallehn, and G. Saake, "Cloud-scale entity resolution: current state and open challenges," *Open Journal of Big Data (OJBD)*, vol. 4, no. 1, pp. 30–51, 2018. (cited on Page 5, 10, 43, and 44)

[13] P. Christen, *Data matching: concepts and techniques for record linkage, entity resolution, and duplicate detection*. Springer Science & Business Media, 2012. (cited on Page 5, 10, 43, and 44)

[14] A. Labrinidis and H. V. Jagadish, "Challenges and opportunities with big data," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 2032–2033, 2012. (cited on Page 1 and 11)

[15] O. Shahmirzadi, A. Lugowski, and K. A. Younge, "Text similarity in vector space models: A comparative study," *Available at SSRN 3259971*, 2018. (cited on Page 2, 9, 10, 34, and 50)

[16] B. Kulis and K. Grauman, "Kernelized locality-sensitive hashing for scalable image search.," in *ICCV*, vol. 9, pp. 2130–2137, 2009. (cited on Page 2 and 38)

[17] Y. Gong, S. Lazebnik, A. Gordo, and F. Perronnin, "Iterative quantization: A procrustean approach to learning binary codes for large-scale image retrieval," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 12, pp. 2916–2929, 2012. (cited on Page 38 and 40)

[18] W. Kong and W.-J. Li, "Isotropic hashing," in *Advances in neural information processing systems*, pp. 1646–1654, 2012. (cited on Page 10, 38, 40, 50, and 90)

[19] W. Liu, J. Wang, R. Ji, Y.-G. Jiang, and S.-F. Chang, "Supervised hashing with kernels," in *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2074–2081, IEEE, 2012. (cited on Page 40)

[20] M. Rastegari, J. Choi, S. Fakhraei, D. Hal, and L. Davis, "Predictable dual-view hashing," in *International Conference on Machine Learning*, pp. 1328–1336, 2013. (cited on Page)

[21] K. He, F. Wen, and J. Sun, "K-means hashing: An affinity-preserving quantization method for learning binary compact codes," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2938–2945, 2013. (cited on Page)

[22] G. Lin, C. Shen, Q. Shi, A. Van den Hengel, and D. Suter, "Fast supervised hashing with decision trees for high-dimensional data," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1963–1970, 2014. (cited on Page 40)

[23] F. Shen, C. Shen, W. Liu, and H. Tao Shen, "Supervised discrete hashing," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 37–45, 2015. (cited on Page 40)

[24] W.-C. Kang, W.-J. Li, and Z.-H. Zhou, "Column sampling based discrete supervised hashing," in *Thirtieth AAAI conference on artificial intelligence*, 2016. (cited on Page 2, 38, 40, 41, and 55)

[25] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, 2017. (cited on Page 2, 50, 53, and 59)

[26] L. Sun, *Skipping-oriented Data Design for Large-Scale Analytics*. PhD thesis, UC Berkeley, 2017. (cited on Page 3)

[27] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman, "The architecture of scidb," in *International Conference on Scientific and Statistical Database Management*, pp. 1–16, Springer, 2011. (cited on Page 3, 4, 10, 11, 37, and 126)

[28] G. Planthaber, M. Stonebraker, and J. Frew, "Earthdb: scalable analysis of modis data using scidb," in *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*, pp. 11–19, ACM, 2012. (cited on Page 4)

[29] M. Stonebraker, J. Duggan, L. Battle, and O. Papaemmanouil, "Scidb dbms research at mit," *IEEE Data Eng. Bull.*, vol. 36, no. 4, pp. 21–30, 2013. (cited on Page 4, 9, 10, 11, 37, and 126)

[30] S. Sakr, *Big data 2.0 processing systems: a survey*. Springer, 2016. (cited on Page 9, 10, and 12)

[31] B. Köse, S. Eken, and A. Sayar, "Playlist generation via vector representation of songs," in *INNS Conference on Big Data*, pp. 179–185, Springer, 2016. (cited on Page 10 and 11)

[32] A. Svyatkovskiy, K. Imai, M. Kroeger, and Y. Shiraito, "Large-scale text processing pipeline with apache spark," in *2016 IEEE International Conference on Big Data (Big Data)*, pp. 3928–3935, IEEE, 2016. (cited on Page 10 and 11)

[33] F. Hu, C. Yang, Y. Jiang, Y. Li, W. Song, D. Q. Duffy, J. L. Schnase, and T. Lee, "A hierarchical indexing strategy for optimizing apache spark with hdfs to efficiently query big geospatial raster data," *International Journal of Digital Earth*, pp. 1–19, 2018. (cited on Page 10 and 11)

[34] A. G. Pablos, M. Cuadros, and G. Rigau, "A comparison of domain-based word polarity estimation using different word embeddings," in *Proceedings of the tenth international conference on language resources and evaluation (LREC 2016)*, pp. 54–60, 2016. (cited on Page 10 and 11)

[35] M. Appel, F. Lahn, W. Buytaert, and E. Pebesma, "Open and scalable analytics of large earth observation datasets: From scenes to multidimensional arrays using scidb and gdal," *ISPRS journal of photogrammetry and remote sensing*, vol. 138, pp. 47–56, 2018. (cited on Page 10 and 11)

[36] R. Guerraoui, E. L. Merrer, R. Patra, and J.-R. Vigouroux, "Sequences, items and latent links: Recommendation with consumed item packs," *arXiv preprint arXiv:1711.06100*, 2017. (cited on Page 10 and 11)

[37] A. Oussous, F.-Z. Benjelloun, A. A. Lahcen, and S. Belfkih, "Big data technologies: A survey," *Journal of King Saud University-Computer and Information Sciences*, vol. 30, no. 4, pp. 431–448, 2018. (cited on Page 10 and 12)

[38] F. Bajaber, R. Elshawi, O. Batarfi, A. Altalhi, A. Barnawi, and S. Sakr, "Big data 2.0 processing systems: Taxonomy and open challenges," *Journal of Grid Computing*, vol. 14, no. 3, pp. 379–405, 2016. (cited on Page 10 and 12)

[39] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015. (cited on Page 10, 15, and 32)

[40] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system.," in *OSDI*, vol. 14, pp. 571–582, 2014. (cited on Page 10 and 19)

[41] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: a timely dataflow system," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 439–455, ACM, 2013. (cited on Page 10 and 14)

[42] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 135–146, ACM, 2010. (cited on Page 10, 13, 14, and 19)

[43] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990. (cited on Page 10 and 12)

[44] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008. (cited on Page 10, 13, and 20)

[45] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pp. 2–2, USENIX Association, 2012. (cited on Page 10, 13, and 32)

[46] A. Gounaris and J. Torres, "A methodology for spark parameter tuning," *Big data research*, vol. 11, pp. 22–32, 2018. (cited on Page 10 and 14)

[47] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *ACM Sigmod Record*, vol. 34, no. 4, pp. 42–47, 2005. (cited on Page 10 and 15)

[48] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, "Stream: The stanford data stream management system," in *Data Stream Management*, pp. 317–336, Springer, 2016. (cited on Page 10 and 15)

[49] A. Arasu, S. Babu, and J. Widom, "The cql continuous query language: semantic foundations and query execution," *The VLDB Journal*, vol. 15, no. 2, pp. 121–142, 2006. (cited on Page 10 and 15)

[50] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, *et al.*, "Storm@ twitter," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 147–156, ACM, 2014. (cited on Page 10 and 15)

[51] J. Kreps, N. Narkhede, J. Rao, *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, pp. 1–7, 2011. (cited on Page 10 and 15)

[52] M. Rodrigues, M. Y. Santos, and J. Bernardino, "Describing and comparing big data querying tools," in *World Conference on Information Systems and Technologies*, pp. 115–124, Springer, 2017. (cited on Page 10 and 17)

[53] M. Rodrigues, M. Y. Santos, and J. Bernardino, "Big data processing tools: An experimental performance evaluation," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 9, no. 2, p. e1297, 2019. (cited on Page 10 and 17)

[54] M. Rodrigues, M. Y. Santos, and J. Bernardino, "Experimental evaluation of big data analytical tools," in *European, Mediterranean, and Middle Eastern Conference on Information Systems*, pp. 121–127, Springer, 2018. (cited on Page 10 and 17)

[55] V. Aluko and S. Sakr, "Big sql systems: an experimental evaluation," *Cluster Computing*, pp. 1–31. (cited on Page 10 and 17)

[56] F. Li, B. C. Ooi, M. T. Özsu, and S. Wu, "Distributed data management using mapreduce," *ACM Computing Surveys (CSUR)*, vol. 46, no. 3, p. 31, 2014. (cited on Page 10 and 17)

[57] T. Lipcon, D. Alves, D. Burkert, J.-D. Cryans, A. Dembo, M. Percy, S. Rus, D. Wang, M. Bertozzi, C. P. McCabe, *et al.*, "Kudu: storage for fast analytics on fast data," *Retrieved June from http://getkudu. io/kudu. pdf. Pages,, and*, 2015. (cited on Page 10 and 18)

[58] A. Floratou, U. F. Minhas, and F. Özcan, "Sql-on-hadoop: full circle back to shared-nothing database architectures," *Proceedings of the VLDB Endowment*, vol. 7, no. 12, pp. 1295–1306, 2014. (cited on Page 10, 17, and 18)

[59] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 1099–1110, ACM, 2008. (cited on Page 10 and 17)

[60] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009. (cited on Page 10 and 17)

[61] R. Yadav, *Spark Cookbook*. Packt Publishing Ltd, 2015. (cited on Page 10 and 21)

[62] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, *et al.*, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pp. 1383–1394, ACM, 2015. (cited on Page 10, 20, 21, 22, and 23)

[63] E. Begoli, J. Camacho-Rodríguez, J. Hyde, M. J. Mior, and D. Lemire, "Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources," in *Proceedings of the 2018 International Conference on Management of Data*, pp. 221–230, ACM, 2018. (cited on Page 10 and 22)

[64] Y. Li, M. Li, L. Ding, and M. Interlandi, "Rios: Runtime integrated optimizer for spark," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 275–287, ACM, 2018. (cited on Page 10 and 22)

[65] G. M. Essertel, R. Y. Tahboub, J. M. Decker, K. J. Brown, K. Olukotun, and T. Rompf, "Flare: Native compilation for heterogeneous workloads in apache spark," *arXiv preprint arXiv:1703.08219*, 2017. (cited on Page 10 and 22)

[66] Y. Zhou, "Large scale distributed file system survey," *Indiana University Bloomington*, 2013. (cited on Page 10 and 23)

[67] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," 2003. (cited on Page 10, 23, and 24)

[68] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008. (cited on Page 10, 23, and 26)

[69] C. Jia and H. Li, *Virtual Distributed File System: Alluxio*, pp. 1–6. Cham: Springer International Publishing, 2018. (cited on Page 10, 20, 26, and 27)

[70] G. Pang and H. Li, *Caching for SQL-on-Hadoop*, pp. 1–5. Cham: Springer International Publishing, 2018. (cited on Page 10, 26, and 27)

[71] D. Barberis, G. Dimitrov, M. Mineev, L. Canali, E. Alexandrov, E. Gallas, J. Sánchez, F. Prokoshin, A. Iakovlev, C. Garcia Montoro, *et al.*, "The atlas eventindex and its evolution based on apache kudu storage," tech. rep., ATL-COM-SOFT-2018-115, 2018. (cited on Page 10, 26, and 28)

[72] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, *et al.*, "The case for ramclouds: scalable high-performance storage entirely in dram," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 4, pp. 92–105, 2010. (cited on Page 10, 27, and 28)

[73] D. Plase, L. Niedrite, and R. Taranovs, "A comparison of hdfs compact data formats: Avro versus parquet/hdfs glaustųjų duomenų formatų palyginimas: Avro prieš parquet," *Mokslas–Lietuvos ateitis/Science–Future of Lithuania*, vol. 9, no. 3, pp. 267–276, 2017. (cited on Page 10 and 29)

[74] A. Floratou, *Columnar Storage Formats*, pp. 1–6. Cham: Springer International Publishing, 2018. (cited on Page 10 and 31)

[75] A. Trivedi, P. Stuedi, J. Pfefferle, A. Schuepbach, and B. Metzler, "Albis: High-performance file format for big data systems," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pp. 615–630, 2018. (cited on Page 10, 29, and 30)

[76] D. Oleś and Z. Nowak, "The performance analysis of distributed storage systems used in scalable web systems," in *International Conference on Information Systems Architecture and Technology*, pp. 287–298, Springer, 2018. (cited on Page 10 and 31)

[77] P. Pirzadeh, M. Carey, and T. Westmann, "A performance study of big data analytics platforms," in *2017 IEEE International Conference on Big Data (Big Data)*, pp. 2911–2920, IEEE, 2017. (cited on Page 10 and 31)

[78] Z. Baranowski, L. Canali, R. Toebbicke, J. Hrivnac, and D. Barberis, "A study of data representation in hadoop to optimize data storage and search performance for the atlas eventindex," in *Journal of Physics: Conference Series*, vol. 898, p. 062020, IOP Publishing, 2017. (cited on Page 10 and 31)

[79] B. Vaddeman, *Data Formats*, pp. 201–208. Berkeley, CA: Apress, 2016. (cited on Page 10 and 29)

[80] S. Bisoyi, P. Mishra, and S. N. Mishra, "Relational query optimization technique using space efficient file formats of hadoop for the big data warehouse system," *Indian Journal of Science and Technology*, vol. 10, pp. 1–7, 02 2017. (cited on Page 10 and 29)

[81] T. Ivanov, T. Rabl, M. Poess, A. Queralt, J. Poelman, N. Poggi, and J. Buell, "Big data benchmark compendium," in *Technology Conference on Performance Evaluation and Benchmarking*, pp. 135–155, Springer, 2015. (cited on Page 10 and 32)

[82] C. Baru, M. Bhandarkar, R. Nambiar, M. Poess, and T. Rabl, "Benchmarking big data systems and the bigdata top100 list," *Big Data*, vol. 1, no. 1, pp. 60–64, 2013. (cited on Page 10 and 32)

[83] S. Wang, I. Pandis, I. Emam, D. Johnson, F. Guitton, A. Oehmichen, and Y. Guo, "Dsimbench: A benchmark for microarray data using r," in *Workshop on Big Data*

Benchmarks, Performance Optimization, and Emerging Hardware, pp. 47–56, Springer, 2014. (cited on Page 10 and 33)

[84] R. Taft, M. Vartak, N. R. Satish, N. Sundaram, S. Madden, and M. Stonebraker, "Genbase: A complex analytics genomics benchmark," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 177–188, ACM, 2014. (cited on Page 10 and 33)

[85] T. Ivanov and S. Izberovic, "Evaluating hadoop clusters with tpcx-hs," *arXiv preprint arXiv:1509.03486*, 2015. (cited on Page 10 and 47)

[86] C. Baru, M. Bhandarkar, C. Curino, M. Danisch, M. Frank, B. Gowda, H.-A. Jacobsen, H. Jie, D. Kumar, R. Nambiar, *et al.*, "Discussion of bigbench: a proposed industry standard performance benchmark for big data," in *Technology Conference on Performance Evaluation and Benchmarking*, pp. 44–63, Springer, 2014. (cited on Page 10 and 47)

[87] A. J. Awan, M. Brorsson, V. Vlassov, and E. Ayguade, "Architectural impact on performance of in-memory data analytics: Apache spark case study," *arXiv preprint arXiv:1604.08484*, 2016. (cited on Page 10 and 33)

[88] D. García-Gil, S. Ramírez-Gallego, S. García, and F. Herrera, "A comparison on scalability for batch big data processing on apache spark and apache flink," *Big Data Analytics*, vol. 2, no. 1, p. 1, 2017. (cited on Page 10 and 32)

[89] A. Fernández, S. García, M. Galar, R. C. Prati, B. Krawczyk, and F. Herrera, *Imbalanced Classification for Big Data*, pp. 327–349. Cham: Springer International Publishing, 2018. (cited on Page 10 and 35)

[90] R. Bordawekar and O. Shmueli, "Exploiting latent information in relational databases via word embedding and application to degrees of disclosure," 2019. (cited on Page 10 and 34)

[91] J. L. Neves and R. Bordawekar, "Demonstrating ai-enabled sql queries over relational data using a cognitive database," *Knowledge Discovery and Data Mining*, 2018. (cited on Page 10 and 34)

[92] R. Bordawekar and O. Shmueli, "Using word embedding to enable semantic queries in relational databases," in *Proceedings of the 1st Workshop on Data Management for End-to-End Machine Learning*, p. 5, ACM, 2017. (cited on Page 10 and 34)

[93] M. Günther, "Freddy: Fast word embeddings in database systems," in *Proceedings of the 2018 International Conference on Management of Data*, pp. 1817–1819, ACM, 2018. (cited on Page 10 and 34)

[94] M. Hamilton, S. Raghunathan, I. Matiach, A. Schonhoffer, A. Raman, E. Barzilay, M. Thigpen, K. Rajendran, J. S. Mahajan, C. Cochrane, *et al.*, "Mmlspark: Unifying machine learning ecosystems at massive scales," *arXiv preprint arXiv:1810.08744*, 2018. (cited on Page 10 and 34)

[95] H. Luu, *Machine Learning with Spark*, pp. 327–383. Berkeley, CA: Apress, 2018. (cited on Page 10 and 35)

[96] D. Agrawal, A. Butt, K. Doshi, J.-L. Larriba-Pey, M. Li, F. R. Reiss, F. Raab, B. Schiefer, T. Suzumura, and Y. Xia, "Sparkbench–a spark performance testing suite," in *Technology Conference on Performance Evaluation and Benchmarking*, pp. 26–44, Springer, 2015. (cited on Page 10 and 36)

[97] J. G. Shanahan and L. Dai, "Large scale distributed data science using apache spark," in *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*, pp. 2323–2324, ACM, 2015. (cited on Page 10)

[98] J. Veiga, R. R. Expósito, X. C. Pardo, G. L. Taboada, and J. Tourifio, "Performance evaluation of big data frameworks for large-scale data analytics," in *2016 IEEE International Conference on Big Data (Big Data)*, pp. 424–431, IEEE, 2016. (cited on Page 10 and 36)

[99] P. Svensson, P. Boncz, M. Ivanova, M. Kersten, N. Nes, and D. Rotem, "Emerging database systems in support of scientific data," *Scientific Data Management: Challenges Technology and Deployment*, pp. 235–277, 2010. (cited on Page 10 and 37)

[100] S. Papadopoulos, K. Datta, S. Madden, and T. Mattson, "The tiledb array data storage manager," *Proceedings of the VLDB Endowment*, vol. 10, no. 4, pp. 349–360, 2016. (cited on Page 10 and 37)

[101] L. Argerich, J. T. Zaffaroni, and M. J. Cano, "Hash2vec, feature hashing for word embeddings," *arXiv preprint arXiv:1608.08940*, 2016. (cited on Page 10)

[102] R. F. Munir, A. Abelló, O. Romero, M. Thiele, and W. Lehner, "Atun-hl: Auto tuning of hybrid layouts using workload and data characteristics," in *European Conference on Advances in Databases and Information Systems*, pp. 200–215, Springer, 2018. (cited on Page 10 and 38)

[103] M. Tang, Y. Yu, W. G. Aref, Q. M. Malluhi, and M. Ouzzani, "Efficient parallel skyline query processing for high-dimensional data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 10, pp. 1838–1851, 2018. (cited on Page 10 and 37)

[104] B. Braams, *Predicate Pushdown in Parquet and Apache Spark*. PhD thesis, Universiteit van Amsterdam, 2018. (cited on Page 10 and 38)

[105] X. Yao and G. Li, "Big spatial vector data management: a review," *Big Earth Data*, vol. 2, no. 1, pp. 108–129, 2018. (cited on Page 10 and 37)

[106] A. Gionis, P. Indyk, R. Motwani, *et al.*, "Similarity search in high dimensions via hashing," in *Vldb*, vol. 99, pp. 518–529, 1999. (cited on Page 10, 38, 39, and 50)

[107] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Communications of the ACM*, vol. 51, no. 1, p. 117, 2008. (cited on Page 10, 38, 39, and 50)

[108] J. Wang, W. Liu, S. Kumar, and S.-F. Chang, "Learning to hash for indexing big data—a survey," *Proceedings of the IEEE*, vol. 104, no. 1, pp. 34–57, 2015. (cited on Page 10, 38, 40, 50, and 90)

[109] Q. Li, Z. Sun, R. He, and T. Tan, "Deep supervised discrete hashing," in *Advances in neural information processing systems*, pp. 2482–2491, 2017. (cited on Page 10 and 42)

[110] P. Zhang, W. Zhang, W.-J. Li, and M. Guo, "Supervised hashing with latent factor models," in *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, pp. 173–182, ACM, 2014. (cited on Page 10, 40, 41, 54, and 55)

[111] Y. Li and T. Yang, "Word embedding for understanding natural language: a survey," in *Guide to Big Data Applications*, pp. 83–104, Springer, 2018. (cited on Page 10)

[112] P. Goyal and E. Ferrara, "Graph embedding techniques, applications, and performance: A survey," *Knowledge-Based Systems*, vol. 151, pp. 78–94, 2018. (cited on Page 10)

[113] O. Barkan and N. Koenigstein, "Item2vec: neural item embedding for collaborative filtering," in *2016 IEEE 26th International Workshop on Machine Learning for Signal Processing (MLSP)*, pp. 1–6, IEEE, 2016. (cited on Page 11)

[114] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013. (cited on Page 11)

[115] B. Furht and F. Villanustre, "Introduction to big data," in *Big data technologies and applications*, pp. 3–11, Springer, 2016. (cited on Page 12)

[116] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: efficient iterative data processing on large clusters," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 285–296, 2010. (cited on Page 14)

[117] M. Garofalakis, J. Gehrke, and R. Rastogi, *Data Stream Management: Processing High-Speed Data Streams*. Springer, 2016. (cited on Page 15)

[118] S. Babu and J. Widom, "Streamon: an adaptive engine for stream query processing," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pp. 931–932, ACM, 2004. (cited on Page 15)

[119] I. Gog, M. Schwarzkopf, N. Crooks, M. P. Grosvenor, A. Clement, and S. Hand, "Musketeer: all for one, one for all in data processing systems," in *Proceedings of the Tenth European Conference on Computer Systems*, p. 2, ACM, 2015. (cited on Page 16)

[120] S. Palkar, J. J. Thomas, A. Shanbhag, D. Narayanan, H. Pirk, M. Schwarzkopf, S. Amarasinghe, M. Zaharia, and S. InfoLab, "Weld: A common runtime for high performance data analytics," in *Conference on Innovative Data Systems Research (CIDR)*, 2017. (cited on Page 16)

[121] R. O. Nambiar and M. Poess, "The making of tpc-ds," in *Proceedings of the 32nd international conference on Very large data bases*, pp. 1049–1058, VLDB Endowment, 2006. (cited on Page 17)

[122] P. Cao, B. Gowda, S. Lakshmi, C. Narasimhadevara, P. Nguyen, J. Poelman, M. Poess, and T. Rabl, "From bigbench to tpcx-bb: Standardization of a big data benchmark," in *Technology Conference on Performance Evaluation and Benchmarking*, pp. 24–44, Springer, 2016. (cited on Page 17)

[123] L. A. Barroso, J. Dean, and U. Hölzle, "Web search for a planet: The google cluster architecture," *IEEE micro*, no. 2, pp. 22–28, 2003. (cited on Page 19)

[124] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*, p. 5, ACM, 2013. (cited on Page 20)

[125] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center.," in *NSDI*, vol. 11, pp. 22–22, 2011. (cited on Page 20)

[126] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pp. 423–438, ACM, 2013. (cited on Page 20)

[127] J. E. Gonzalez, "From graphs to tables the design of scalable systems for graph analytics," in *Proceedings of the 23rd International Conference on World Wide Web*, pp. 1149–1150, ACM, 2014. (cited on Page 20)

[128] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: A resilient distributed graph system on spark," in *First International Workshop on Graph Data Management Experiences and Systems*, p. 2, ACM, 2013. (cited on Page 20)

[129] S. Venkataraman, Z. Yang, D. Liu, E. Liang, H. Falaki, X. Meng, R. Xin, A. Ghodsi, M. Franklin, I. Stoica, *et al.*, "Sparkr: Scaling r programs with spark," in *Proceedings of the 2016 International Conference on Management of Data*, pp. 1099–1104, ACM, 2016. (cited on Page 20)

[130] H. Lee, K. Brown, A. Sujeeth, H. Chafi, T. Rompf, M. Odersky, and K. Olukotun, "Implementing domain-specific languages for heterogeneous parallel computing," *Ieee Micro*, vol. 31, no. 5, pp. 42–53, 2011. (cited on Page 22)

[131] D. Borthakur, "The hadoop distributed file system: Architecture and design," *Hadoop Project Website*, vol. 11, no. 2007, p. 21, 2007. (cited on Page 24)

[132] A. Verma and S. Venkataraman, "Efficient metadata management for cloud computing applications," tech. rep., 2010. (cited on Page 24)

[133] D. Fesehaye, R. Malik, and K. Nahrstedt, "Edfs: a semi-centralized efficient distributed file system," in *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, p. 28, Springer-Verlag, 2009. (cited on Page 24)

[134] R. T. Kaushik and M. Bhandarkar, "Greenhdfs: towards an energy-conserving, storage-efficient, hybrid hadoop compute cluster," in *Proceedings of the USENIX annual technical conference*, vol. 109, p. 34, 2010. (cited on Page 24)

[135] F. B. Schmuck and R. L. Haskin, "Gpfs: A shared-disk file system for large computing clusters.," in *FAST*, vol. 2, 2002. (cited on Page 25)

[136] S. R. Soltis, T. M. Ruwart, and M. T. OKeefe, "The global file system," 1996. (cited on Page 25)

[137] Y. Gu and R. L. Grossman, "Sector and sphere: the design and implementation of a high-performance data cloud," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 367, no. 1897, pp. 2429–2445, 2009. (cited on Page 25)

[138] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th symposium on Operating systems design and implementation*, pp. 335–350, USENIX Association, 2006. (cited on Page 26)

[139] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970. (cited on Page 26)

[140] A. Floratou, N. Megiddo, N. Potti, F. Özcan, U. Kale, and J. Schmitz-Hermes, "Adaptive caching in big sql using the hdfs cache," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pp. 321–333, ACM, 2016. (cited on Page 27)

[141] D. Barberis, S. C. Zárate, J. Cranshaw, A. Favareto, Á. F. Casaní, E. Gallas, C. Glasman, S. G. De La Hoz, J. Hřivnáč, D. Malon, *et al.*, "The atlas eventindex: architecture, design choices, deployment and first operation experience," in *Journal of Physics: Conference Series*, vol. 664, p. 042003, IOP Publishing, 2015. (cited on Page 27)

[142] A. Collaboration, "The atlas experiment at the cern large hadron collider jinst 3," *S08003*, pp. 1–437, 2008. (cited on Page 28)

[143] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu, "Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems," in *2011 IEEE 27th International Conference on Data Engineering*, pp. 1199–1208, IEEE, 2011. (cited on Page 30)

[144] A. Floratou, J. M. Patel, E. J. Shekita, and S. Tata, "Column-oriented storage techniques for mapreduce," *Proceedings of the VLDB Endowment*, vol. 4, no. 7, pp. 419–429, 2011. (cited on Page 30)

[145] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. Borkar, Y. Bu, M. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, *et al.*, "Asterixdb: A scalable, open source bdms," *Proceedings of the VLDB Endowment*, vol. 7, no. 14, pp. 1905–1916, 2014. (cited on Page 32)

[146] C. Baru, M. Bhandarkar, R. Nambiar, M. Poess, and T. Rabl, "Setting the direction for big data benchmark standards," in *Technology Conference on Performance Evaluation and Benchmarking*, pp. 197–208, Springer, 2012. (cited on Page 47)

[147] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen, "Bigbench: towards an industry standard benchmark for big data analytics," in *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, pp. 1197–1208, ACM, 2013. (cited on Page 47)

[148] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, *et al.*, "Bigdatabench: A big data benchmark suite from internet services," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 488–499, IEEE, 2014. (cited on Page 33 and 47)

[149] M. Kunjir, P. Kalmegh, and S. Babu, "Thoth: Towards managing a multi-system cluster," *Proceedings of the VLDB Endowment*, vol. 7, no. 13, pp. 1689–1692, 2014. (cited on Page 47)

[150] C. Luo, J. Zhan, Z. Jia, L. Wang, G. Lu, L. Zhang, C.-Z. Xu, and N. Sun, "Cloudrank-d: benchmarking and ranking cloud computing systems for data processing applications," *Frontiers of Computer Science*, vol. 6, no. 4, pp. 347–362, 2012. (cited on Page 47)

[151] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *ACM SIGPLAN Notices*, vol. 47, pp. 37–48, ACM, 2012. (cited on Page 47)

[152] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pp. 41–51, IEEE, 2010. (cited on Page 33 and 47)

[153] K. Kim, K. Jeon, H. Han, S.-g. Kim, H. Jung, and H. Y. Yeom, "Mrbench: A benchmark for mapreduce framework," in *2008 14th IEEE International Conference on Parallel and Distributed Systems*, pp. 11–18, IEEE, 2008. (cited on Page 47)

[154] A. Sangroya, D. Serrano, and S. Bouchenak, "Mrbs: A comprehensive mapreduce benchmark suite," *LIG, Grenoble, France, Research Report RR-LIG-024*, 2012. (cited on Page 47)

[155] A. Sangroya, D. Serrano, and S. Bouchenak, "Mrbs: Towards dependability benchmarking for hadoop mapreduce," in *European Conference on Parallel Processing*, pp. 3–12, Springer, 2012. (cited on Page 47)

[156] 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 *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pp. 165–178, ACM, 2009. (cited on Page 47)

[157] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin, "Mapreduce and parallel dbmss: friends or foes?," *Communications of the ACM*, vol. 53, no. 1, pp. 64–71, 2010. (cited on Page 47)

[158] J. Ferrarons, M. Adhana, C. Colmenares, S. Pietrowska, F. Bentayeb, and J. Darmont, "Primeball: A parallel processing framework benchmark for big data applications in the cloud," in *Technology Conference on Performance Evaluation and Benchmarking*, pp. 109–124, Springer, 2013. (cited on Page 47)

[159] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura, "Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, p. 53, ACM, 2015. (cited on Page 47)

[160] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "The case for evaluating mapreduce performance using workload suites," in *2011 IEEE 19th annual international symposium on modelling, analysis, and simulation of computer and telecommunication systems*, pp. 390–399, IEEE, 2011. (cited on Page 47)

[161] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1802–1813, 2012. (cited on Page 47)

[162] R. Nambiar, M. Poess, A. Dey, P. Cao, T. Magdon-Ismail, A. Bond, *et al.*, "Introducing tpcx-hs: the first industry standard for benchmarking big data systems," in *Technology Conference on Performance Evaluation and Benchmarking*, pp. 1–12, Springer, 2014. (cited on Page 47)

[163] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 143–154, ACM, 2010. (cited on Page 47)

[164] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi, "Ycsb++: benchmarking and performance debugging advanced features in scalable table stores," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, p. 9, ACM, 2011. (cited on Page 47)

[165] G. Brown, A. Pocock, M.-J. Zhao, and M. Luján, "Conditional likelihood maximisation: a unifying framework for information theoretic feature selection," *Journal of machine learning research*, vol. 13, no. Jan, pp. 27–66, 2012. (cited on Page 33)

[166] R. Lu, G. Wu, B. Xie, and J. Hu, "Stream bench: Towards benchmarking modern distributed stream computing frameworks," in *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pp. 69–78, IEEE, 2014. (cited on Page 33)

[167] S. Perera and S. Suhothayan, "Solution patterns for realtime streaming analytics," in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, pp. 247–255, ACM, 2015. (cited on Page 33)

[168] T.-W. Chang, "Binding of cells to matrixes of distinct antibodies coated on solid surface," *Journal of immunological methods*, vol. 65, no. 1-2, pp. 217–223, 1983. (cited on Page 33)

[169] O. Alter, P. O. Brown, and D. Botstein, "Singular value decomposition for genome-wide expression data processing and modeling," *Proceedings of the National Academy of Sciences*, vol. 97, no. 18, pp. 10101–10106, 2000. (cited on Page 33)

[170] R. Bordawekar, B. Bandyopadhyay, and O. Shmueli, "Cognitive database: A step towards endowing relational databases with artificial intelligence capabilities," *arXiv preprint arXiv:1712.07199*, 2017. (cited on Page 34)

[171] O. Levy and Y. Goldberg, "Linguistic regularities in sparse and explicit word representations," in *Proceedings of the eighteenth conference on computational natural language learning*, pp. 171–180, 2014. (cited on Page 34)

[172] T. Mikolov, Q. V. Le, and I. Sutskever, "Exploiting similarities among languages for machine translation," *arXiv preprint arXiv:1309.4168*, 2013. (cited on Page)

[173] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, pp. 3111–3119, 2013. (cited on Page 34)

[174] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 1532–1543, 2014. (cited on Page 34)

[175] A. Fernández, S. del Río, N. V. Chawla, and F. Herrera, "An insight into imbalanced big data classification: outcomes and challenges," *Complex & Intelligent Systems*, vol. 3, pp. 105–120, Jun 2017. (cited on Page 35)

[176] M. Zaharia, A. Chen, A. Davidson, A. Ghodsi, S. A. Hong, A. Konwinski, S. Murching, T. Nykodym, P. Ogilvie, M. Parkhe, *et al.*, "Accelerating the machine learning lifecycle with mlflow," *Data Engineering*, p. 39, 2018. (cited on Page 36)

[177] S. Borzsony, D. Kossmann, and K. Stocker, "The skyline operator," in *Proceedings 17th international conference on data engineering*, pp. 421–430, IEEE, 2001. (cited on Page 37)

[178] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "Progressive skyline computation in database systems," *ACM Transactions on Database Systems (TODS)*, vol. 30, no. 1, pp. 41–82, 2005. (cited on Page)

[179] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, "Skyline with presorting," in *ICDE*, vol. 3, pp. 717–719, 2003. (cited on Page)

[180] D. Kossmann, F. Ramsak, and S. Rost, "Shooting stars in the sky: An online algorithm for skyline queries," in *Proceedings of the 28th international conference on Very Large Data Bases*, pp. 275–286, VLDB Endowment, 2002. (cited on Page)

[181] K. C. Lee, B. Zheng, H. Li, and W.-C. Lee, "Approaching the skyline in z order," in *Proceedings of the 33rd international conference on Very large data bases*, pp. 279–290, VLDB Endowment, 2007. (cited on Page 37)

[182] S. Shekhar, M. R. Evans, V. Gunturi, K. Yang, and D. C. Cugler, "Benchmarking spatial big data," in *Specifying Big Data Benchmarks*, pp. 81–93, Springer, 2012. (cited on Page 37)

[183] X. Tong, J. Ben, Y. Liu, and Y. Zhang, "Modeling and expression of vector data in the hexagonal discrete global grid system," *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. 4, p. W2, 2013. (cited on Page 37)

[184] A. S. Das, M. Datar, A. Garg, and S. Rajaram, "Google news personalization: scalable online collaborative filtering," in *Proceedings of the 16th international conference on World Wide Web*, pp. 271–280, ACM, 2007. (cited on Page 39)

[185] H. Koga, T. Ishibashi, and T. Watanabe, "Fast agglomerative hierarchical clustering algorithm using locality-sensitive hashing," *Knowledge and Information Systems*, vol. 12, no. 1, pp. 25–53, 2007. (cited on Page 39)

[186] D. Brinza, M. Schultz, G. Tesler, and V. Bafna, "Rapid detection of gene–gene interactions in genome-wide association studies," *Bioinformatics*, vol. 26, no. 22, pp. 2856–2862, 2010. (cited on Page 39)

[187] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pp. 380–388, ACM, 2002. (cited on Page 39)

[188] J. Wang, S. Kumar, and S.-F. Chang, "Semi-supervised hashing for scalable image retrieval," 2010. (cited on Page 40)

[189] M. Norouzi and D. M. Blei, "Minimal loss hashing for compact binary codes," in *Proceedings of the 28th international conference on machine learning (ICML-11)*, pp. 353–360, Citeseer, 2011. (cited on Page 40)

[190] C. Strecha, A. Bronstein, M. Bronstein, and P. Fua, "Ldahash: Improved matching with smaller descriptors," *IEEE transactions on pattern analysis and machine intelligence*, vol. 34, no. 1, pp. 66–78, 2011. (cited on Page 40)

[191] H. Hotelling, "Analysis of a complex of statistical variables into principal components.," *Journal of educational psychology*, vol. 24, no. 6, p. 417, 1933. (cited on Page 40)

[192] Y. Weiss, A. Torralba, and R. Fergus, "Spectral hashing," in *Advances in neural information processing systems*, pp. 1753–1760, 2009. (cited on Page 40)

[193] D. Zhang, J. Wang, D. Cai, and J. Lu, "Self-taught hashing for fast similarity search," in *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, pp. 18–25, ACM, 2010. (cited on Page 40)

[194] W. Liu, J. Wang, S. Kumar, and S.-F. Chang, "Hashing with graphs," 2011. (cited on Page 40)

[195] W. Liu, C. Mu, S. Kumar, and S.-F. Chang, "Discrete graph hashing," in *Advances in neural information processing systems*, pp. 3419–3427, 2014. (cited on Page 40)

[196] Q.-Y. Jiang and W.-J. Li, "Scalable graph hashing with feature transformation," in *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015. (cited on Page 40)

[197] J. Song, Y. Yang, Z. Huang, H. T. Shen, and R. Hong, "Multiple feature hashing for real-time large scale near-duplicate video retrieval," in *Proceedings of the 19th ACM international conference on Multimedia*, pp. 423–432, ACM, 2011. (cited on Page 40)

[198] D. Zhang, F. Wang, and L. Si, "Composite hashing with multiple information sources," in *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, pp. 225–234, ACM, 2011. (cited on Page 40)

[199] S. Kumar and R. Udupa, "Learning hash functions for cross-view similarity search," in *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011. (cited on Page 40)

[200] Y. Zhen and D.-Y. Yeung, "A probabilistic model for multimodal hash function learning," in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 940–948, ACM, 2012. (cited on Page 40)

[201] Y. Zhen and D.-Y. Yeung, "Co-regularized hashing for multimodal data," in *Advances in neural information processing systems*, pp. 1376–1384, 2012. (cited on Page 40)

[202] J. Song, Y. Yang, Y. Yang, Z. Huang, and H. T. Shen, "Inter-media hashing for large-scale retrieval from heterogeneous data sources," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pp. 785–796, ACM, 2013. (cited on Page 40)

[203] M. Ou, P. Cui, F. Wang, J. Wang, W. Zhu, and S. Yang, "Comparing apples to oranges: a scalable solution with heterogeneous hashing," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 230–238, ACM, 2013. (cited on Page 40)

[204] D. Zhang and W.-J. Li, "Large-scale supervised multimodal hashing with semantic correlation maximization," in *Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014. (cited on Page 40)

[205] G. Ding, Y. Guo, and J. Zhou, "Collective matrix factorization hashing for multimodal data," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2075–2082, 2014. (cited on Page 40)

[206] B. Wu, Q. Yang, W.-S. Zheng, Y. Wang, and J. Wang, "Quantized correlation hashing for fast cross-modal search," in *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015. (cited on Page 40)

[207] Z. Lin, G. Ding, M. Hu, and J. Wang, "Semantics-preserving hashing for cross-view retrieval," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3864–3872, 2015. (cited on Page 40)

[208] M. Norouzi, D. J. Fleet, and R. R. Salakhutdinov, "Hamming distance metric learning," in *Advances in neural information processing systems*, pp. 1061–1069, 2012. (cited on Page 40)

[209] J. Wang, J. Wang, N. Yu, and S. Li, "Order preserving hashing for approximate nearest neighbor search," in *Proceedings of the 21st ACM international conference on Multimedia*, pp. 133–142, ACM, 2013. (cited on Page 40)

[210] J. Wang, W. Liu, A. X. Sun, and Y.-G. Jiang, "Learning hash codes with listwise supervision," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 3032–3039, 2013. (cited on Page 40)

[211] Q. Wang, Z. Zhang, and L. Si, "Ranking preserving hashing for fast similarity search," in *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015. (cited on Page 40)

[212] R. Xia, Y. Pan, H. Lai, C. Liu, and S. Yan, "Supervised hashing for image retrieval via image representation learning," in *Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014. (cited on Page 41)

[213] H. Lai, Y. Pan, Y. Liu, and S. Yan, "Simultaneous feature learning and hash coding with deep neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3270–3278, 2015. (cited on Page 41)

[214] F. Zhao, Y. Huang, L. Wang, and T. Tan, "Deep semantic ranking based hashing for multi-label image retrieval," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1556–1564, 2015. (cited on Page 41)

[215] R. Zhang, L. Lin, R. Zhang, W. Zuo, and L. Zhang, "Bit-scalable deep hashing with regularized similarity learning for image retrieval and person re-identification," *IEEE Transactions on Image Processing*, vol. 24, no. 12, pp. 4766–4779, 2015. (cited on Page 41)

[216] V. Erin Liong, J. Lu, G. Wang, P. Moulin, and J. Zhou, "Deep hashing for compact binary codes learning," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2475–2483, 2015. (cited on Page 41)

[217] M.-M. Deza and E. Deza, *Dictionary of distances*. Elsevier, 2006. (cited on Page 43, 60, 65, and 91)

[218] F. Gorunescu, *Data Mining: Concepts, models and techniques*, vol. 12. Springer Science & Business Media, 2011. (cited on Page 60)

[219] A. Huang, "Similarity measures for text document clustering," in *Proceedings of the sixth new zealand computer science research student conference (NZCSRSC2008), Christchurch, New Zealand*, vol. 4, pp. 9–56, 2008. (cited on Page 43, 65, and 91)

[220] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pp. 785–794, ACM, 2016. (cited on Page 43, 59, 65, and 91)

[221] A. Nagrani, S. Albanie, and A. Zisserman, "Learnable pins: Cross-modal embeddings for person identity," in *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 71–88, 2018. (cited on Page 50)

[222] S. J. Moran, "Learning to hash for large scale image retrieval," 2016. (cited on Page 50)

[223] M. Vlachos, *Indexing and Similarity Search*, pp. 1438–1442. Boston, MA: Springer US, 2009. (cited on Page 50)

[224] D. Xu, T. J. Cham, S. Yan, L. Duan, and S.-F. Chang, "Near duplicate identification with spatially aligned pyramid matching," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 20, no. 8, pp. 1068–1079, 2010. (cited on Page 50)

[225] C. Doersch, S. Singh, A. Gupta, J. Sivic, and A. A. Efros, "What makes paris look like paris?," *Communications of the ACM*, vol. 58, no. 12, pp. 103–110, 2015. (cited on Page 50)

[226] C. E. Yoon, O. O'Reilly, K. J. Bergen, and G. C. Beroza, "Earthquake detection through computationally efficient similarity search," *Science Advances*, vol. 1, no. 11, 2015. (cited on Page 50)

[227] H. Köpcke, A. Thor, and E. Rahm, "Evaluation of entity resolution approaches on real-world match problems," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 484–493, 2010. (cited on Page 59 and 125)

[228] S. Das, A. Doan, P. S. G. C., C. Gokhale, and P. Konda, "The magellan data repository." https://sites.google.com/site/anhaidgroup/projects/data. (cited on Page 59)

[229] K. Sugawara, H. Kobayashi, and M. Iwasaki, "On approximately searching for similar word embeddings," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, vol. 1, pp. 2265–2275, 2016. (cited on Page 64)

[230] L. R. Baxter, R. Baxter, P. Christen, *et al.*, "A comparison of fast blocking methods for record," 2003. (cited on Page 125)

[231] A. Saeedi, E. Peukert, and E. Rahm, "Comparative evaluation of distributed clustering schemes for multi-source entity resolution," in *European Conference on Advances in Databases and Information Systems*, pp. 278–293, Springer, 2017. (cited on Page 125)

[232] M. Ebraheem, S. Thirumuruganathan, S. Joty, M. Ouzzani, and N. Tang, "Distributed representations of tuples for entity resolution," *Proceedings of the VLDB Endowment*, vol. 11, no. 11, pp. 1454–1467, 2018. (cited on Page 126)

[233] S. Mudgal, H. Li, T. Rekatsinas, A. Doan, Y. Park, G. Krishnan, R. Deep, E. Arcaute, and V. Raghavendra, "Deep learning for entity matching: A design space exploration," in *Proceedings of the 2018 International Conference on Management of Data*, pp. 19–34, ACM, 2018. (cited on Page 126)

[234] X. Chen, G. Campero Durand, R. Zoun, D. Broneske, Y. Li, and G. Saake, "The best of both worlds: Combining hand-tuned and word-embedding-based similarity measures for entity resolution," *BTW 2019*, 2019. (cited on Page 126)

[235] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus," *IEEE Transactions on Big Data*, 2019. (cited on Page 126 and 128)