AN EMPIRICAL COMPARATIVE ANALYSIS OF SKYLINE QUERY ALGORITHMS FOR INCOMPLETE DATA

By

Anthony Messana, B.S in Computer Science

A thesis submitted to the Graduate Committee of

Ramapo College of New Jersey in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Spring, 2025

Committee Members:

Ali Al-Juboori, Advisor

Sourav Dutta, Reader

Lawrence D'Antonio, Reader

COPYRIGHT

Table of Contents

List of Tables

List of Figures

ABSTRACT

Skyline queries are a popular and useful technique for multi-criteria analysis, but the presence of incomplete data complicates the retrieval of the skyline. Namely, incompleteness introduces the problems of *intransitivity* and *cyclic dominance*. Over the years, many algorithms have been developed to find skylines over incomplete data by addressing the two aforementioned problems. For software engineers working on Big Data applications or for researchers interested in the incomplete Skyline problem, it can be useful to know in which contexts a particular class of algorithm may perform best. We sought to investigate the differing approaches to dealing with the unique challenges of computing the skyline over incomplete data. We picked three recently developed algorithms to represent certain classes of incomplete skyline algorithms, and benchmarked them in different contexts. We controlled for the correlation of the dataset, the size of the dataset, and the dimensionality of the dataset. The three algorithms, PFSIDS (Liu et. tal), TSI (He et. al) and BTIS(Yuan et. al) represent sorting-based, table-scan based, and bucket-based approaches respectively. We found that the sorting-based algorithm performed the best in general, except in the case of high dimensional anti-correlated data. The table-scan based algorithm was observed to work best in small, higher-dimensional datasets and its performance did not change significantly with respect to the correlation of the data. The bucket-based approach generally performed the worst, which we believe to be due to the overhead of initializing the possibly large number of buckets for each data class.

CHAPTER ONE

INTRODUCTION

1.1 Overview

Skyline queries are a popular method for supporting multi-criteria analysis in a dataset (Tiakas et. al; Choi et al). The skyline of a dataset is a set of tuples such that none of the tuples are dominated by any other tuple in the dataset (Chomicki et. al, 2013; Borzsonyi et. al). We say tuple t_1 dominates tuple t_2 when t_1 is as good as t_2 across all dimensions, and is better in at least one (Borzsonyi et. al). What defines "as good" and "better" is dependent on the dominance relation defined between tuples. The dominance relation can be any one that defines an ordering among the values of each tuple's dimensions, but is typically either a less than or equal to, or greater than or equal to relationship. A dominance relation with this structure is known as Pareto dominance (Chomicki et. al, 2013). When all pareto-dominated tuples of a dataset are removed, the result is the Pareto frontier (i.e, the skyline). The traditional skyline query problem assumes a complete dataset, meaning that all tuples have values present for all of their dimensions. When this holds true, the transitivity of the dominance relationship can be exploited to prune many tuples out of the search for those in the skyline.

A typical example of an application of the skyline query is a hotel recommendation system. A user may desire to find hotels that are both the cheapest and closest to the beach, which present a multi-criteria optimization problem. A skyline query can return to the user the set of hotels that represent the optimal trade-offs between price and location. Suppose we have the set of tuples {(5, 3), (7, 2), (7, 3), (6, 4), (5, 8)} where the first value is the hotel's price, and the second is the hotel's distance from the beach. In this scenario, the dominance relation used to compare tuples would be

less than or equal to, as we want to minimize both cost and distance. The resulting skyline would consist of hotels that may only be worse along one dimension, but better along another dimension. In this scenario, the tuples making up our skyline are {(5, 3), (7, 2)}. We can see that {(7, 3), (6,4), and (5, 8)} are all dominated by (5, 3). Tuple (5,3) is better than (7,3) in price, and is equally good in distance, is better than (6, 4) along all dimensions, and is better than (5,8) in distance and equally good in price. The only tuple not dominated by (5, 3) is (7, 2), as (7, 2) is worse in price but better in distance.

Understanding traditional skyline algorithms is still relevant to our discussion of the incomplete-case, as many incomplete skyline algorithms process the incomplete data in such a way that allows for the use of traditional skyline algorithms to be employed (Khalefa et. al; Yuan et al; Dehaki et. al; Rahman, Hasan). The most basic approach is the nested loop, or NL, which naively calculates the skyline by comparing every object with every other object in the dataset via a nested loop (Borzsonyi et. al). Another version is the Block-Nested Loop, more practicable for database applications, where a window is maintained that holds some limited number of records. The BNL has a worst-case performance of O (n^2), leading to the development of more efficient variations of BNL such as the Sort-Filter Skyline algorithm (SFS) that topologically sorts the data beforehand, leading to a worst-case performance of O ($ln + n \log n$), where l is the size of the skyline (Chomicki et. al, 2003). The Divide and Conquer algorithm (DC) computes the skyline through recursive partitioning of the dataset (Borszonyi et. al). The median value of a dimension is calculated to partition the dataset into two partitions P_1 and P_2. Sub-skylines for each partition are calculated by further dividing each partition into smaller partitions until they consist of one, or a few objects. The sub-skylines are then merged and dominated tuples are removed from the merged skylines. This merging continues until all sub-skylines are merged into one skyline for the

entire dataset. DC has a worst case time complexity of O ($n(\log n)^{(d-2)}$) + O ($n \log n$), where d is the dimensionality of the data. (Tiakas et al; Borzsony et. al).

An important class of traditional skyline algorithms are those that utilize a spatial index, typically in the form of an R-tree (Tiakas et al; Luo et al;). The basic idea behind these algorithms involves traversing an R-tree in a depth-first-search manner and using a set of candidate nodes to prune entire branches away from the search space. Two examples of such algorithms are the Nearest Neighbor search (NN), and the Branch and Bound Algorithm (BBS) (Papadias et. al; Kossman et al). In the NN algorithm, all data objects are indexed an R-tree, and a suitable distance measure is chosen for the Nearest Neighbor search. (e.g, Euclidean distance in two-dimensional data). The next nearest neighbor is found iteratively from a defined origin point, and some region of the R-tree can be pruned as each neighbor is visited. The BBS algorithm expands upon NN by introducing the use of a heap and the defined distance measure is the L_1 norm of the data vectors. Both NN and BBS are I/O efficient and progressively construct the skyline as they run, which can be useful if live reporting of the skyline is preferred (Tiakas et. al).

1.2 Problem Statement

There are many cases in which a dataset will have missing values, making comparisons between tuples across their different dimensions difficult. “Big Data” and Machine learning applications dealing with millions of data points and IoT systems that receive data from possibly unreliable devices are some common examples where missing data is common (Adhikari et. al; Emmanuel et. al). In a dataset with missing values, comparisons are not guaranteed to be *transitive*, and *cyclic dominance* is possible. The three main categories of incomplete data skyline algorithms are replace-based (Khalefa et. al), sorting-based (Bharuka, Kumar), and bucket-based approaches (Lee

et al), all of which use different methods to bring back *transitivity* to skyline comparisons and eliminate the problem of *cyclic dominance*.

A replace-based approach seeks to replace missing values with some value, making the incomplete dataset a complete dataset. A traditional skyline algorithm is then applied to the transformed dataset. The value replacing the incomplete dimension is typically a prediction of what that value should be, based on inferred dependencies between attributes (Alwan et. al). Another proposed approach involves replacing the missing value with a sentinel value, such as negative infinity, and then applying a traditional skyline algorithm on the transformed dataset (Khalefa et. al). The resulting skyline on this transformed dataset is a superset of the skyline. The actual skyline is calculated by reverting the replaced dimensions back to incomplete dimensions in the computed skyline, and then doing pairwise comparisons among all of the tuples in the computed skyline to filter out dominated tuples.

Bucket-based approaches group tuples into buckets based on their bitmap representation (Khalefa et. al; Lee et. al; Dehatki et al). The bitmap representation of a tuple is a bit-string where 1's represents a complete dimension and 0's represents incomplete dimensions. Thus, a bucket is created for every possible combination of complete and incomplete dimensions that exist within the data set. There can be at most 2^d of these buckets, where d is the number of dimensions. The problems of intransitivity and *cyclic dominance* are not an issue for tuples within a bucket, as they all have the same complete dimensions. This allows for a traditional skyline algorithm to be used to calculate local skylines on each bucket. These local skylines are then merged into a candidate skyline, and then a final pairwise comparison is done among all tuples in the candidate skyline to calculate the final skyline.

Sort-based algorithms such as SIDS typically create an index of tuples through the creation of sorted lists (Bharuka, Kumar). In SIDS for example, sorted lists of tuples for each dimension is created, with the idea that tuples at the front of the list have a high likelihood of pruning many non-skyline tuples. If a tuple is incomplete on some dimension *d*, is it not present in the sorted list for that particular dimension. Each of the lists are accessed in a round robin fashion, and so tuples with the best values across each dimension are accessed on every pass through of the lists in an attempt to prune as many tuples as possible.

What we seek to accomplish is a comparative study of three state-of-the-art skyline algorithms for incomplete data. We have chosen one bucket-based (Yuan et. al), one sorting-based (Liu et. al), and one table-scan based algorithm (He et. al) to compare with each other. To our best knowledge, there does not exist any empirical comparative studies of recent skyline algorithms for incomplete data. We specifically chose these algorithms due to their recency and differing approaches in solving the issues of intransitivity, cyclical dominance, and search space pruning. Typical performance metrics for skyline algorithms will be used, such as number of comparisons, I/O usage, and execution time.

1.3 Research Questions

This section breaks down the questions we seek to answer in this research study.

1. What is the impact of sorting versus bucketing as a strategy to deal with incomplete data?
2. What is the impact of the size, dimensionality and correlation of the data on the performance of each algorithm?
3. What are the limitations of the selected skyline algorithms when applied to datasets with missing values?

4. What are the key characteristics and operational principles of the selected skyline algorithms for incomplete datasets?
5. What are the differences between bucket-based, sort-based, and table-scan-based skyline algorithms in terms of algorithmic design and handling of missing values?
6. What conclusions can be drawn from the empirical comparison, and what recommendations can be made for selecting a skyline algorithm based on the characteristics of the input data?

1.4 Scope of Research

The scope of this research is focused on skyline algorithms over incomplete data. The selected algorithms that are compared are designed for batch-processing a dataset as opposed to stream-processing incomplete data in real time. Skyline algorithms for dynamic incomplete datasets are discussed as part of the literature review of this research study. Top-k and k-dominant queries over incomplete data are also discussed in the literature review section. The datasets used to compare each algorithm include small, medium and large datasets from both synthetic and real sources. Synthetic datasets include independent, correlated, and anticorrelated data, and the real data contain NBA, insurance, and movie review data. All of the algorithms in question assume totally numeric data, and thus the real and synthetic data only contain numeric values.

1.5 Significance of the Study

The primary significance of this thesis is a comparative study of the performance of three well-known skyline algorithms designed to operate on datasets with incomplete data. Many modern applications increasingly deal with data from possibly unreliable sources, such as IoT sensors or

crowd-sourced data, so the need to work with incomplete data becomes more prominent. This study offers valuable insights for software engineers and system designers working on a system with incomplete datasets highlighting the strengths and the limitations of each algorithm under various conditions. For instance, if a table has a large number of attributes (dimensions), they may choose to implement the skyline algorithm that handles high-dimensional data the most efficiently. Lastly, beyond its practical applications, this study can also serve as a reference point for future researchers on skyline queries over incomplete data, contributing to a deeper understanding of algorithmic trade-offs and guiding the development of more efficient solutions in this evolving field.

1.6 Organization of the Thesis

Chapter 1 is an introduction to the topic of skyline queries and the problem that incomplete data poses. Some background information, research objectives, and the significance of the thesis are provided in this chapter.

Chapter 2 provides definitions needed to understand the fundamental concepts regarding skyline queries over incomplete data. This chapter will also contain the literature review, looking at previously proposed algorithms for the problem, as well as an explanation of the three algorithms used in the comparative study.

Chapter 3 outlines the methodology of the research. It will detail the metrics used to measure each algorithm's performance, the specifications of the environment each benchmark is run on, the language used to implement the algorithms, and the data sets that will be used for the benchmarks.

Chapter 4 is the analysis and discussion of the results of the benchmarks. It discusses the results of each algorithm, and contains figures and graphs that compare each algorithm along a specific metric. It will detail which contexts each algorithm appears to do better, and which ones it struggles with in comparison to the others.

Chapter 5 is the conclusion, which recommends when it may be appropriate to use any of the three algorithms. We consider the results of the analysis and discussion, and also consider some qualitative factors such as ease of implementation.

CHAPTER TWO

BACKGROUND

2.1 Introduction

This chapter presents the fundamental concepts needed to understand the problems with calculating the skyline on incomplete data, as well as understanding the previous work on the topic. Mathematical definitions of a skyline, dominance, and an incomplete dataset have been given. Definitions for key concepts regarding incomplete data skyline algorithms are explained, such as the bitmap representation of tuples, bucketing, virtual points, and shadow skylines. A discussion of previous work is also included in this chapter, detailing their approaches to handling incomplete data. An explanation of the three chosen algorithms, Priority-First Sort-Based Incomplete Data Skyline (PFSIDS), Table-Scan based Skyline over Incomplete Data (TSI) and Multidimensional Incomplete Data based on Classification tree (BTIS) is also given in this chapter.

2.2 Preliminaries and Definitions

We will denote the incomplete dataset as D, which consists of n tuples where each tuple t consists of m dimensions. The dimensions of some tuples, $a_1,\ldots,a_m$, may be incomplete, which will be denoted by a '-'. For example, the tuple (1, -, 3) has an incomplete second dimension. The values for each dimension are assumed to be numerical in all of the algorithms presented in this chapter. We will define the dominance relationship to prefer greater values, although this has no bearing on how any of the algorithms work. We will also introduce the definitions of bucketing, shadow skylines, optimal virtual points, as these are key concepts used in the previous work section and in the algorithms benchmarked in our thesis (Yuan et. al).

Definition 1 (*Skyline*) Given a dataset *D*, the skyline of *D* consists of a subset S of *D* such that none of the elements in *S* are dominated by any of the elements in *D*.

Definition 2 (*Dominance Relation*) The dominance relation between two tuples t_1 and t_2 in some dataset *D* with *m* dimensions, where t_1 is said to dominate t_2, holds when $t_1 \neq t_2$ and $\forall_i$, $t_1[i] \geq t_2[i]$. In other words, t_1 dominates t_2 when t_1 is not worse than t_2 in any dimension, and is at least better than t_2 in one dimension.

Definition 3 (*Bitmap and Bucketing*) Let a bucket *N* consists of a set of tuples who share the same bitmap encoding. The bitmap encoding of a tuple is an *m* length bitstring, where *m* is the number of dimensions, that has a 1 in the *i*-th position if the *i*-th dimension is complete, and 0 otherwise.

Definition 4 (*Virtual Point*) Given a point *p* in the local skyline of a bucket N_i, if *p* dominates a point *q* in bucket N_j, then point *p* may be used as a virtual point.

Definition 5 (*Optimal Virtual Point*) An optimal virtual point $E = (e_1, e_2, \ldots e_m)$ of a bucket consists of the optimal value of each dimension among tuples in the local skyline. The value of each dimension is labeled with the original source point.

Definition 6 (*Shadow Skyline*) The shadow skyline is the result of filtering out points dominated by the optimal virtual point and its source points from a local skyline.

2.3 Overview of Algorithms

In this section, we present an overview of three state-of-the-art skyline algorithms that are designed specifically to deal with incomplete data. These algorithms address the challenges posed by missing attribute values when computing skylines. The first algorithm, BTIS (Yuan et. al), uses a weighted classification tree and optimal virtual points to efficiently compute skylines. The second, PFIDS (Liu et. al), builds on sort-based indexing and introduces a priority-based processing mechanism to improve skyline computation. The third, TSI (He et. al), adopts a sequential table-scan approach optimized for large-scale datasets and proposes pruning enhancements to improve

performance. The following subsections provide detailed descriptions of each algorithm's design and implementation.

2.3.1 Skyline Query on Incomplete Data based on Classification Tree (BTIS)

The first incomplete skyline algorithm that we will discuss is BTIS which was introduced by Yuan et. al. This work proposes an algorithm for skylines on incomplete data that utilizes a weighted classification tree. The incomplete data set is classified using this tree, and it serves as a basis for the second step of the skyline query algorithm. The incomplete data-weighted classification tree is introduced to address the problems of data redundancy, low data classification efficiency and slow classification speed. Separate dimensions of incomplete data are separated in intermediate nodes, and each dimension is assigned a weight based on missing values. Classification is achieved by horizontal indexing of leaf nodes, which is guided by the weight values of each dimension.

An algorithm for calculating a skyline in multidimensional incomplete data is proposed to deal with data redundancy and the presence of large amounts of useless data, which can both reduce query efficiency. It uses the classification tree and the concept of optimal virtual points to achieve this. An optimal point is a point that contains a maximum value for a dimension from a local skyline. Optimal virtual points can rapidly identify points with high dominating potential, which allows for greater point pruning and fewer comparisons. The incomplete data-weighted classification tree is a B+ tree like structure. The root node of a classification tree stores the dimensional values of data tuples to be classified. The dimensions are sequentially written from 1 to n into the second layer's leaf nodes. Missing dimensions are given a weight of 0, and complete dimensions a weight of 1. The data on each dimension is stored in the leaf nodes of the third layer,

and data is extracted through horizontal indexing and then classified by the dimension's weights. The classified data is then assigned to their respective classes.

The algorithm for creating the weighted classification tree consists of two stages. In the first stage, a random training set consisting of 70% of the original data is created. This training set must contain all classification scenarios to ensure the accuracy of the training tree. The root of the tree contains all of the data tuples, and the intermediate branch nodes contain the dimension attributes of the tuples. The intermediate nodes branch based on the completeness of its dimension attribute. Incomplete attributes branch to a leaf node with a weight of 0, and complete ones to a node with a weight of 1. In the second stage, a horizontal index accesses the varying weights of each dimension, and the tuple is classified based on these weights. Finally, the classified tuple is placed into the bucket corresponding to its classification. Leaf nodes are then emptied, and the processes repeated with the next tuple until all data is classified.

Skyline queries are calculated in each bucket, with the remaining points being the local skyline. The optimal virtual point is then calculated for each bucket, by picking the maximum value for each dimension among all tuples in the local skyline. The locus point, the points from which maximum dimension values are derived, are maintained for each virtual point Vi. The virtual points are added to each local skyline, resulting in the creation of a shadow skyline and a candidate skyline for each bucket. The candidate skyline are points not dominated by the virtual point's constituent points. The candidate skyline points are then compared to the points in the shadow skyline, and if any are not dominated by any shadow points, they remain in the candidate skyline. Finally, the candidate skyline points are compared with one another to produce the final global skyline.

2.3.2 Priority-Based Skyline Query Processing for Incomplete Data (PFSIDS)

In the work introduced by Liu et. al, an algorithm based on SIDS is proposed. In the Sort-Based Incomplete Data Skyline (SIDS) algorithm, *d* lists are created for each dimension of the data set, where each list is sorted on one dimension. A list is picked as a starting point, and then each point in the list is compared to one another to remove dominated tuples. The algorithm moves to another list and repeats the process in a round-robin fashion, visiting the first list again after completing the last list. If a tuple has been processed k times, where k is the number of complete dimensions a tuple has, it is determined to be part of the skyline. Drawbacks of this approach are that it ignores the fact that the dominance of a point is also dependent on the number of complete dimensions it has, and that the addition of new data is costly because all of the lists would need to be remade and resorted.

In the PFIDS (Liu et. al) algorithm, an index is created based on the cumulative number of complete dimensions and the points sorted order across a dimension. All of the points with i complete dimensions are aggregate and placed into a list Li. is created for each combination of cumulative complete dimensions. Li consists of d sorted arrays, where each array contains points with i complete dimensions sorted on some dimension d. Finally, each of the lists Li are put in another list in ascending order of i, completing the index.

The next stage uses the index to create the skyline. A CandidateSet data structure is initialized, and at first contains the entire dataset D, and the skyline is initialized to null. Points with multiple complete dimensions are in multiple arrays within a list L_i and so a variable processedCount for each point is maintained to count how many times it has been processed. If a point has been processed the same number of times as the number of its complete dimensions, it can be made part of the skyline. The lists of the index are accessed in a round-robin fashion, and

points are iteratively processed according to a position pointer. A point is processed by comparing to every other point currently in the CandidateSet. For example, if the pointer is at 0, then each point at index 0 across all the arrays in a list will be processed. Missing dimensions are skipped when a point is being processed. After the last list is processed, the position pointer is increased by one, and the process repeats until the CandidateSet is empty, meaning all points have either been pruned or added to the Skyline.

2.3.3 Table-scan based Skyline over Incomplete data (TSI)

The final algorithm in question, devised by He & Han, is based on a sequential table scan, and is optimized for massive amounts of data. The basic idea behind TSI is that possible candidate tuples are identified in an initial scan of the table, and then the set of candidates is further refined in another scan to discard dominated tuples. In stage 1, tuples are retrieved sequentially from the table, and a set of candidate tuples is maintained. If the selected tuple dominates anything in the set of candidates, they are removed from the set. Likewise, if the tuple is dominated by anything in the set, it is discarded. In this first scan, intransitivity and cyclic dominance of incomplete data tuples are not considered. This means that the resulting set of candidates is a superset of the skyline, rather than the final skyline.

In the second scan, the tuples are once again accessed sequentially in the table. At each iteration, each tuple in the candidate set is checked to see if it's dominated by the current tuple from the table. Tuples in the candidate set that are dominated by the current tuple are then removed from the set. At the end of this scan, everything left in the candidate set forms the final skyline of the data set.

Another version of the TSI algorithm is proposed that includes a pruning operation in the first stage. A majority of the execution time cost has been analyzed to occur in the first stage of TSI. In the pruning version, a preconstructed data structure helps the first stage skip a large number of dominated tuples. A small number of "pruning tuples" are extracted from the data set, which are expected to dominate a larger number of tuples. Due to the possibly high dimensionality of the data, pruning tuples are chosen in respect to values of a single dimension, and in regards to its number of complete attributes. Tuples with a larger number of incomplete attributes, and greater values in its complete attributes have the potential to prune many points. At the beginning of stage 1, *m* pruning tuples are maintained in a min-heap to keep tuples in order of highest dominance capability. The pruning tuples are used to generate the initial set of candidates in stage 1, rather than needing to scan through the entire table like in the basic version. Stage 1 determines which pruning tuples should be kept in the candidate set to be used in Stage 2, which remains unchanged from the basic version of TSI.

2.4 Previous Work

In this section, we give an overview of other algorithms for the skyline query over incomplete data. These algorithms are designed for a variety of contexts, such as dynamic databases, crowd-sourced data, and parallel systems, but all have in common that they are meant for incomplete data. Some are algorithms for specific cases of the skyline query, such as the top-*k* query, *k*-dominant query, and the skyline-join query. Algorithms in these categories face the same problems of cyclic dominance and intransitivity as the regular skyline query and only differ in their final output. and so are still relevant to the discussion of skyline queries over incomplete data.

2.4.1 Answering Skyline Queries over Incomplete Data with Crowdsourcing

Miao et. al explores a crowdsourcing-based approach to creating skylines on incomplete data. The name of their query framework is called BayesCrowd, and it makes use of Bayesian networks to take into account data correlation. It also uses a c-table (conditional database) to represent objects with incomplete data. In a c-table, objects are paired with a propositional logic formula that, when satisfied, means that object is in the skyline. The probability of this propositional formula being true is the probability that this object is in the skyline. The BayesCrowd framework consists of two phases, modeling and crowdsourcing. The modeling phase involves constructing the c-table, and the crowdsourcing phase consists of three task selection strategies under budget and latency constraints. A marginal utility function is developed to measure the benefit of crowdsourcing a particular task. This function takes into account the difficulty in computing the probability of an object's conditional formula. An adaptive DPLL (Davis-Putnam-Logemann Loveland) algorithm is proposed to speed up this computation. The paper's experiments show that the BayesCrowd framework outperforms existing crowd-sourcing based skyline methods along the metrics of efficiency, cost, and latency.

2.4.2 Computing Skyline Query on Incomplete Data

The work proposed by Rahman & Hasan introduces an algorithm for computing a skyline query on incomplete data that deals with the issues of intransitivity, and cyclic dominance. The algorithm consists of eight phases: counting incompleteness, pruning, finding a weighting factor, creating a weighted matrix, grouping, finding local skylines, finding candidate skylines, and retrieving the final skyline. First, each tuple has the number of incomplete dimensions counted. This number is then used to determine what percent "complete" the tuple is, which is calculated

as a ratio of the number of complete dimensions over the total number of dimensions. Any tuple with a completeness less than 25% is removed from the dataset. Next, a weighting factor is applied to each tuple to smooth the dataset and deprioritize more incomplete objects. The weighting factor for each tuple is computed like so: Wi = (incomplete dimensions of i) + 1. A weighted matrix of the data set is then computed by dividing the values of each dimension by the tuples weighting factor. Tuples with more incomplete dimensions have a greater weight, and thus their values are reduced more than tuples with more complete dimensions. Tuples are then grouped together by their bitmap representation, which denotes the complete dimensions a tuple has. Tuples with the same set of complete dimensions are thus grouped together. In the next stage, local skylines are calculated within these groups. This is possible since all tuples in these groups have the same complete dimensions, so a traditional skyline algorithm can be used to acquire these local skylines. Finally, all of the local skylines are merged into a candidate skyline, and each tuple is compared with one another to create the final skyline result. Experimental results show that this algorithm outperforms the SCSA algorithm, which was the most current skyline approach as of the writing of this thesis.

2.4.3 CrowdSJ Skyline - Join Query Processing of Incomplete Datasets with Crowdsourcing

In the paper by Ding et. al, they created a crowdsourcing-based approach to the skyline-join query, which is a variant of the skyline query that returns the skyline of multiple datasets. The name of their proposed method is CrowdSJ. The problem is divided into two possible scenarios. One is where the skyline-join only involves the unknown crowdsourcing attribute and the join attribute, known as Partial Skyline Join with Crowdsourcing (PSJCrowd). The other is when the

skyline-join involves all attributes, known as the All-Skyline Join with Crowdsourcing (ASJCrowd). In the PSJCrowd case, the known data set is filtered and a level-preference tree index is employed. For ASJCrowd, the database is also filtered, and level-preference tree index is built based on the known attributes of the incomplete. The paper also proposes a third algorithm, Crowd-SJ single, which is simply a crowdsourcing skyline-join algorithm applied on a single database.

2.4.4 Efficient Computation of Skyline Queries Over a Dynamic and Incomplete Database

The work introduced by Dehaki et. al proposes an algorithm for computing the skyline query over incomplete and dynamic databases, called DyIn-Skyline. In a dynamic database, insertions, updates, and deletes invalidate earlier produced skyline results. In order to avoid needing to recompute the skyline over the entire database, the analysis of the dominance relations between data points needs to be preserved. The first phase of the algorithm involves computing the initial skyline over the entire incomplete database. The second phase involves processing skyline queries over a dynamic database. In phase 1, three list data structures are computed to be used to capture dominance relations. These lists are: Domination History (DH), Bucket Dominating (BDG), and Bucket Dominated. These lists can be used in phase 2 to avoid re-examining the entire database to compute a skyline, a dynamic database. Phase 1 uses a bucket-based approach for computing a skyline over an incomplete database, which groups tuples into buckets based on their bitmap representation. This allows tuples to be placed in groups where they are directly comparable, as they will all have the same complete dimensions. During this process, the DH, BDG, and BD data structures are created, which are then used in phase 2.

2.4.5 Efficient k-dominant skyline query over incomplete data using MapReduce

The work by Ding et. al introduces an algorithm for a *k*-dominant skyline query overt an incomplete data set. A *k*-dominant skyline differs from a traditional skyline, as only *k* of the d dimensions are considered during comparison. Changing the parameter *k* can increase the selectivity of what is placed in the skyline, which can be useful in cases where the skyline is large. The algorithm proposed in the paper is based on a dominant-hierarchical tree in a MapReduce environment. The data set is preprocessed via the construction of an Incomplete Data index, based on a dominant hierarchical tree (ID-DHT), that divides data into subspaces where dominance calculations can be made, based on the possible values of *k*. The MapReduce environment leverages distributed computing to improve the efficiency of the data preprocessing.

2.4.6 IDSA: An Efficient Algorithm for Skyline Queries Computation on Dynamic and Incomplete Data with Changing States

The work presented by Gulzar et. al devised an algorithm for skyline queries on dynamic databases with incomplete data, known as the Incomplete Dynamic Skyline Algorithm (IDSA). The IDSA algorithm integrates two key optimization techniques: pruning and selecting superior local skylines. The pruning process identifies new skylines before insert/update operations by leveraging derived skylines, while the selection of superior local skylines further refines the results by eliminating non-skylines. These optimizations significantly reduce the number of domination tests, avoiding the need to recompute skylines for the entire updated database. Extensive experiments on both real and synthetic datasets show that IDSA outperforms existing methods in terms of both the number of domination tests and processing time.

2.4.7 Top-*k* Dominating Queries on Incomplete Data

A top-*k* dominating query is defined as a query which returns k objects that dominate the maximum number of objects in a given dataset. The work presented by Miao et. al is a systematic review of TKD queries on incomplete databases, and proposes efficient algorithms for computing TKD queries on incomplete data. Novel techniques such as upper bound score pruning, bitmap pruning, and partial score pruning to boost query efficiency. The results demonstrate that the new heuristics significantly boost TKD query efficiency.

CHAPTER THREE

METHODOLOGY

3.1 Introduction

In this chapter, we discuss the implementation details of each algorithm, their execution environment, and how each algorithm is benchmarked. The most important aspect of this chapter is the discussion of the different types of datasets that the algorithms are run on. The results will serve to draw conclusions on which contexts an algorithm is better suited for. Variables such as the number of dimensions, missing rate, total number of rows, and correlation between dimensions will be changed to observe their effects. Additional parameters concerning the execution environment will be tweaked to observe the effect of memory usage for each algorithm. The time complexity of each algorithm will be given, as well as pseudocode for each algorithm. Since a functional programming language was used for implementation (Scala), the pseudo code will also resemble a functional style of programming.

3.2 Implementation and Execution Environment

PFSIDS, TSI and BTIS were implemented in Scala 3.6.3 and written in a functional style when possible. This is important to note, as it may increase the memory footprint due to the copying of data structures. Another important implementation detail to note is that the basic version of the TSI version was used, as opposed to the version that utilizes a min-heap to prune out dominated tuples. Each algorithm was run in a single-threaded context and benchmarked using Java Microbenching Harness (JMH). The benchmarks were run in SingleShotTime mode, meaning that test did not have warm-up iterations. The purpose of warm-up iterations is to get a more

realistic run-time result, as the JVM needs to “warm-up” on a particular code path by executing it multiple times to allow for JIT compiler optimization. The run-times collected from the single-shot benchmarks are still useful for the sake of comparing the performance of each algorithm, but it could be a path of future work to determine which algorithms benefit the most from a warm-up period on the JVM. The metrics of interest are execution time, number of comparisons, and memory usage. Memory usage is quantified by the garbage-collector’s allocation rate, which measures the amount of memory allocated in MB per second. Five single shot benchmarks are run consecutively for each dataset tested, and the final results are the average of all runs. The number of comparisons cannot be ascertained from the JMH, so that is tracked through the use of a variable that increments whenever two tuples are checked for dominance. The specifications of the machine executing the algorithm are: AMD Ryzen 5 2600 Six-Core Processor 3.40 GHz, 32 GB DDR4 RAM, AMD RX 580, Windows 10 Home Edition 64 bit. The default JVM heap size of 1GB is used.

3.3 Datasets

Both real and synthetic datasets are used for benchmarking each algorithm. For the synthetic datasets, anti-correlated, correlated, and independent datasets were generated. For each type of data, there is a three, five, seven and nine dimensional dataset. For each dimension, datasets of size 60 KB, 80 KB and 100 KB are created. The three dimensional datasets have a missing rate of 33.33%, the five dimensional a missing rate of 20%, and the 9 dimensional a missing rate of 11%. It's important to note that as the number of dimensions increases as the size of the file stays constant, the number of tuples will decrease. The total number of data processed does not change but the logical size of the input, defined as the number of tuples, will decrease. The real-world

datasets are from NBA player statistics (https://www.nba.com/stats), The Insurance Company Benchmark (COIL 2000) (https://kdd.ics.uci.edu/databases/tic/tic.html), and Movielens movie reviews (https://grouplens.org/datasets/movielens/). The NBA dataset consists of player statistics during the regular season, and eleven, thirteen, and fifteen dimensional versions of the dataset will be used. Each dataset will be the same size at 100 KB. The COIL 2000 dataset contains information about customers at an insurance company. Eleven, thirteen, and fifteen dimensional versions of the data set a size of 100 KB are used. For both the NBA and COIL 2000 datasets, the eleven dimensional dataset has a missing rate of 9.1%, the thirteen dimensional 7.7% and fifteen dimensional 6.7%. Lastly is the Movielens data set consisting of movie review data. Each dataset contains three dimensions and is tested at sizes 300 KB, 600 KB, 900 KB and 1200 KB. All of the Movielens datasets have a missing rate of 33.3%. From this selection of datasets, we can test the effects of correlation, large amounts of data, and high dimensionality on the performance of each algorithm.

3.4 Algorithmic Analysis and Pseudocode

The algorithms in question have different asymptotic time complexity and can expect to scale differently as input size and dimensionality increases. Table 3.1 below shows the worst, average and best case run times of each algorithm, where n is the number of tuples in the dataset and d the number of dimensions. In the case of BTIS, n represents the size of the training set that creates the weighted classification tree, and m is the size of the incomplete data.

Table 3.1 - Asymptotic Complexity of Algorithms

Algorithm	Worst-Case	Average Case	Best Case
PFSIDS (Liu et. al)	O ($dn^2 + dn \log n$)	~ O($n \log n$)	O($dn + dn \log n$)
TSI (He et. al)	O(n^2)	O(n^2)	O(n^2)
BTIS - training classification tree (Yuan et. al)	O ($2^d * n + 2^d * d \log n$)	O ($2^d n + 2^d d \log n$)	O ($2^d n + 2^d d \log n$)
BTIS - classifying missing data (Yuan et. al)	O ($2^d * m + 3d * 2^d \log m$)	O ($2^d * m + 3d * 2^d \log m$)	O ($2^d * m + 3d * 2^d \log m$)

3.4.1 General Constructs Used in Implementation

A *Point* class was created to represent a data point within each dataset. It contains a *dominates* method, which returns true if the first point dominates the second point, and false otherwise. There is also the *completeDims* method, which returns the set of complete dimensions a point has, and the instance variables *numDims*, which holds the number of complete dimensions of a point. Each point object maintaining a count of its complete dimensions slightly changes the implementation of PFSIDS. Originally, a *dimCount* data structure was initialized to represent the number of complete dimensions of each point, but this is now unnecessary. A wrapper around the *dominates* function called *compare* is present in each algorithm's implementation. Whenever the *compare* function is called, a global comparison counter is incremented, and then *dominates* is called on the two points to be checked. This is important to note, as it means the number of comparisons represents the total number of dominance checks, rather than the total number of individual comparisons between the dimensions of points.

Since the algorithms were implemented in a functional manner, when possible, the pseudocode will make use of functional constructions such as map and filter operations. Notice

the use of the **map** construct on lines 3 and 5 in Fig 3.2, for example. The left hand side is the input sequence to be mapped, and the right-hand side is the mapping function that returns what each element is to be mapped to. The output of **map** is a new sequence created by applying the mapping function to each element of the input. The "yield" keyword is used to specify what value the map operation returns for each element. The **where** construct in Fig 3.3 is a filter operation, where the elements of the sequence on the LHS satisfying the predicate on the RHS are returned.

3.4.2 PFSIDS Pseudocode

The PFSIDs algorithm represents the sorting strategy for computing a skyline query over incomplete data. The idea behind PFSIDS is that tuples with fewer complete dimensions have a higher dominance capability than those with more complete dimensions. Tuples with few complete dimensions and optimal values in their complete dimensions are considered likely to prune many non-skyline tuples. The logic is that a tuple with fewer complete dimensions is less likely to be dominated, as it has fewer dimensions it can be compared against. Obviously, having a more optimal value along a certain dimension also makes the tuple more likely to prune other tuples. The sorted index built in PFSIDS thus ranks tuples by the value of a dimension, and its cumulative complete dimensions. Figures 3.2 and 3.3 show the pseudo code of the PFSIDS algorithm, which consists of building the index, and then iterating through the index to prune tuples and build the skyline. Lastly, we define M as the set of dimensions $\{d_1, d_2, \ldots d_m\}$ in the dataset.

On line 3 of Figure 3.1, we map each number of cumulative complete dimensions to a list l_i. List l_i is formed by mapping each dimension d in M to *listPoints* sorted on d. Thus, l_i consists of $|M|$ arrays sorted along a dimension d, and L consists of i lists. Finally, L is sorted by i, the number of cumulative dimensions present in the tuples of each list.

PFSIDS - SortedIndex

```
   Input: dataset D
   Output: list of lists L containing arrays sorted along each dimension
1      L ←
2         cumulative number of complete dimensions across all points i map
3            listPoints ← p ∈ D where |p| = i
4            l_i ← d ∈ M map yield listPoints sorted on d.
5            yield l_i
6       return L sorted on i.
```

Figure 3.1: PFSIDS Sorted Index Pseudocode

PFSIDS - ComputeSkyline

```
   Input:
      list of lists L containing arrays sorted along each dimension
      candidateSet initialized as the entire dataset D
      skyline initialized as empty set
      integer counter i
      integer counter pos
   Output: Skyline set S
1      if candidateSet = ∅ then return Skyline
2      else
3         points ← d ∈ M map yield L[i][d][pos]
4         candidateSet ← c ∈ candidateSet where ∄p ∈ points, p dominates c
5       dominatedPoints ← p ∈ points where ∃ c ∈ candidateSet,  c dominates p
6         candidateSet ← candidateSet diff dominatedPoints
7         for p in points do
8            processedCount[p] ← processedCount[p] + 1
9       skylinePoints ← p ∈ points where
                              candidateSet contains p and p.numDims = processedCount[p]
10        candidateSet ←  candidateSet diff skylinePoints
11      skyline ← skyline union skylinePoints
12        if i = L.length then ComputeSkyline(L, candidateSet, skyline, 0, pos + 1)
13        else ComputeSkyline(L, candidateSet, skyline, i + 1, pos)
```

Figure 3.2: PFSIDS Skyline Computation Pseudocode

Our implementation differs from the original PFSIDS in that it is done tail-recursively rather than through iteration, but it achieves the same result in the same manner. Since we are using a functional language that uses immutable data structures by default, this was not amenable

to the iterative approach which mutates data structures. While this may seem inefficient, Scala makes key optimizations that greatly reduce the overhead of recursion and mutable data structures. Tail-recursion optimization prevents a new stack-frame from being allocated on each recursive call and achieves recursion in a way that mimics regular iteration. Scala utilizes persistent data structures to reduce the overhead of creating new copies of immutable data structures. When a new copy is made, it shares as much of the existing structure as possible with the old version. For example, if we have a list l_1 and create a new list l_2 by prepending a value to l_2, l_2 will have a new head, but then the rest of l_2 will share its structure with l_1.

PFSIDS - Main Function

Input: list of lists *L* containing arrays sorted along each dimension
Output: Skyline set S

```
1    processedCount[p] ← 0 for all points p in D
2    candidateSet ← D
3    skyline ← ∅
4    return ComputeSkyline(L, candidateSet, skyline, 0, 0)
```

Figure 3.3: The PFSIDS Main Function

Line 3 of the ComputeSkyline function selects the pruning points from index *L* by mapping each dimension to a point from *L*. We select list l_i from *L*, the array sorted along dimension *d* from l_i, and the point at index *pos* from the sorted array. At the first iteration, we would grab points from l_1, where each point has only one complete dimension. Each list l_i contains arrays sorted along each dimension. Thus on the first iteration, when *pos* equals 0, we are picking the points with one cumulative dimension that have the best value in a particular dimension *d*. In the next iteration, we pick the best points from l_2, which have two cumulative points. This continues until the end of *L* is reached, and then the variable *i* is set to 0 and *pos* is incremented to 1 (Lines 11-12). When *pos*

equals 1, the second-best points are picked with respect to each dimension from each list. This process will repeat until all points are removed from *candidateSet*.

Line 5 is a filter operation that keeps points from *candidateSet* only if they are not dominated by any element in *points*. It's also possible that the points we selected to prune *candidateSet* are also dominated by a point in *candidateSet*. Line 6 filters for any pruning tuples that are dominated. In line 7 the **diff** operation removes from *candidateSet* the points in *dominatedPoints* to eliminate dominated pruning tuples. Line 9 updates *processedCount*, which keeps track of how many times a pruning tuple has been selected from the index. Since a point can appear multiple times in the index, we do not want to process it more times than the point has complete dimensions. If a point *p* has been processed *p.numDims* times and is still in *candidateSet*, it is removed from *candidateSet* (line 11) and promoted to the skyline (line 12).

The ComputeSkyline function will continue to recur until *candidateSet* is eventually empty, and the conditional expression at line 2 returns the skyline.

3.4.3 TSI (Basic) Pseudocode

The TSI algorithm represents a sequential table-scan approach to handling incomplete data. TSI appears to be unique in that it does not employ an index or use some bucketing strategy to solve intransitivity and cyclic dominance. TSI consists of two sequential passes over the dataset as shown in Figure 3.4. The first pass builds a set of possible candidates, and the second pass discards candidates that are dominated. The first pass starts with an empty candidate set, and as tuples are retrieved from the dataset, it is checked against the candidate set to see if either it dominates any candidates, or is dominated by a candidate. If the retrieved tuple is not dominated, it is added to the set, and any candidates dominated by the tuple are removed from the set. In the second pass, tuples are retrieved from the dataset and each retrieved tuple is checked against each

member of the candidate set to see if it is dominated. At the end of the second pass, whatever is left in the candidate set is the final skyline.

TSI (Basic)

```
   Input: dataset D
   Output: Skyline set S
1     candidates ← ∅
2     for p in D do
3        if candidates = ∅ then candidates ← candidates union p
4        else candidates ← c ∈ candidates where ¬ ( p dominates c )
5        if ∄c ∈ candidates, c dominates p then candidates ← candidates union p
6
7     for p in D do
8        candidates ← c ∈ candidates where ¬ ( p dominates c )
9     return candidates
```

Figure 3.4: Pseudocode for TSI (Basic)

He et. al provides a simple proof demonstrating that two sequential scans are sufficient for dealing with intransitivity and cyclic dominance, and will return the correct result. First, they propose that after the first scan, *candidates* will contain a superset of the skyline set. For every tuple in the dataset, if it is part of the skyline, it will definitely be in the candidate set at the end of the first pass. If a tuple t_1 is not part of the skyline, there exists some other tuple t_2 that dominates it. If t_1 occurs prior to t_2 in the dataset, t_2 will remove t_1 from the candidate set. However, if t_2 occurs prior to t_1, it is possible some other tuple dominates t_2 before t_1 is reached, and t_1 remains in the candidate set. On the second pass, every tuple in the dataset is again retrieved to see if it dominates something in the candidate set. This means that t_2 will have a chance to dominate and remove t_1 from the candidate set. After all dominated tuples are removed, the tuples that are left represent the skyline.

3.4.4 BTIS Pseudocode

BTIS is the representative algorithm for bucket-based approaches for incomplete data. A novel aspect of BTIS is that it utilizes a weighted classification tree to categories and bucket data. The structure of this classification tree is similar to that of a B+ tree (Yuan et. al), and consists of a root node *N* that stores a tuple. Let *k* be the number of dimensions of the dataset, then our tree has *k* internal nodes, which are direct children of the root. Each internal node has two leaf nodes, one with a weight of one and the other with a weight of zero. For a tuple stored in the root node, its values are sequentially written to the internal layer's leaf nodes, left to right. For each dimension of a tuple, it will branch to the left leaf node with weight 0 if the dimension is missing, and branch to the right node with a weight of 1 if it is present. The leaf now contains the tuple's data, and the resulting weights from the tree are used to classify the tuple. Using the combination of weights at each leaf node, the tuple is exported to a bucket corresponding to its classification. A classification is represented by a *k*-length bit vector, where a value of 1 on the *i*-th position means the ith dimension is complete, and is 0 otherwise. The data is cleared from the leaf nodes, and the next tuple is placed in the root node, repeating the process until all data is classified. Figure 3.5 illustrates the steps of the BTIS algorithm.

Once each tuple is placed into a bucket for its respective class, local skylines are calculated within each bucket. Since all tuples within the bucket have the same complete and missing dimensions, the issues of intransitivity and cyclic dominance are avoided. Once a local skyline is calculated for each bucket, the optimal virtual point is calculated for each bucket. The optimal virtual point consists of the optimal value for each dimension among all tuples within a bucket. The optimal virtual point also maintains the original source points from which the optimal dimension values come from. The optimal virtual point of each bucket is introduced into every

other bucket, as long as the virtual point shares one common dimension with the tuples in the bucket. Any points in a local skyline dominated by an optimal virtual point and by one of its source points are placed in the shadow skyline. Points not dominated by an optimal virtual point are placed in the candidate skyline. Any candidate skyline points of a bucket that are dominated by shadow points in other buckets are removed from the shadow skyline. Finally, the points of the candidate skyline are compared with one another to determine if any of them dominate each other, and dominated tuples are removed. The tuples left in the candidate skyline after this represents the complete skyline.

BTIS - Weighted Classification Tree

```
   Input: Training sample set T, dataset D
   Output: Classified datasets C1, C2, … , Cn
1  N ← empty root node
2  N.children ← k empty internal nodes
3  buckets ← { }
4  for c in N.children do
5      c.leftChild ← empty leaf node with weight 0
6      c.rightChild ← empty leaf node with weight 1
7  for t in T do
8     N.data ← t
9     for i in 0..k do
10        N.children[i].data ← t[i]
11    for c in N.children do
12        if c.data is missing then
13            c.leftChild.data ← c.data
14        else c.rightChild.data ← c.data
15    classVector ← k-length vector of weights from leaf nodes
16    buckets[classVector] ← t
17 for d in D do
18    use classification tree to place each tuple d into corresponding bucket
19 return buckets /* the contents of the buckets are our classification sets C1, C2, …, Cn */
```

Figure 3.5: BTIS Weighted Classification Tree Pseudocode

Lines 1-16 in Figure 3.5 represent the data modeling phase of the classification process. The sample dataset *T* is 70% of the size of the original data, and should contain tuples that represent

all possible cases of combinations of missing dimensions, so that every possible data class can be modeled. Lines 1-6 initialize the classification tree and the *buckets* data structure, which may be implemented as a hashmap or some other form of index. Lines 7-16 are responsible for uncovering all of the different types of classes of tuples that exist in the data. On lines 15-16, a vector is created from the weights of the leaf nodes, and then a new class is created in the *bucket* data structure from the vector. Lines 17-18 represent a similar process to Lines 7-16, except tuples are placed into the existing classes created in the first part, and the tree classifies the entire dataset rather than the sample set.

BTIS - Skyline Computation

```
   Input: Classification Datasets C1, C2, … Cn  /* The buckets variable */
   Output: Skyline set S
1  localSkylines ← (class, b) ∈ buckets map yield (class, p ∈ b where ∄q ∈ b, q dominates p)
2  optimalVPs ← (class, b) ∈ buckets map yield (class, optimalVP of bucket b)
3  shadowSkyline ← (classi , localSky) ∈ localSkylines
4                       flatmap
5                          vps ← (classj, (vp, sources)) ∈ optimalVPs where (classj ⋀ classi ≠
0)
6                          yield p ∈ localSky where
                                    ∃ (class, (vp, sources)) ∈ vps,  vp dominates p
                                    ⋀ ∃ s ∈ sources, s dominates p )
7  candSkyline ← ((class, localSky) ∈ localSkylines flatmap yield localSky)
8                      diff shadowSkyline
9  globalSkyline ← p ∈ candSkyline where
10                            ∄q ∈ candSkyline, q dominates p ⋀ ∄s ∈ shadowSkyline, s
dominates p
11  return globalSkyline
```

Figure 3.6: BTIS Skyline Computation Pseudocode

Figure 3.6 demonstrates the steps of BTIS skyline computation process. sOn line 3, we create *localSkyline*, a hashmap containing the local skyline of each class. The mapping operation selects the points from each bucket that are not dominated by any other point in the bucket. At line 4, we

calculate the optimal virtual point of each bucket, creating another hashmap associating a data class (our *k-length* bit vector) with the optimal virtual point of that class' bucket. The optimal virtual point as mentioned earlier, consists of the optimal value along all dimensions of a bucket, and also contains the original source points where those optimal values came from. On line 5, we use the **flatmap** operation, which flattens the result of a map operation by a single level. On line 9, we compute the candidate skyline by flattening the localSkyline hashmap into a sequence of points, and then using the **diff** operation to remove any local skyline points also in the shadow skyline. However, since candidate skyline points could still be dominated by either another point in the candidate set and a shadow skyline point, we need to do additional dominance checks. Candidate points dominated by another candidate point or by a shadow point are filtered out on line 11, and the result is the complete global skyline.

CHAPTER FOUR

ANALYSIS AND DISCUSSION

4.1 Introduction

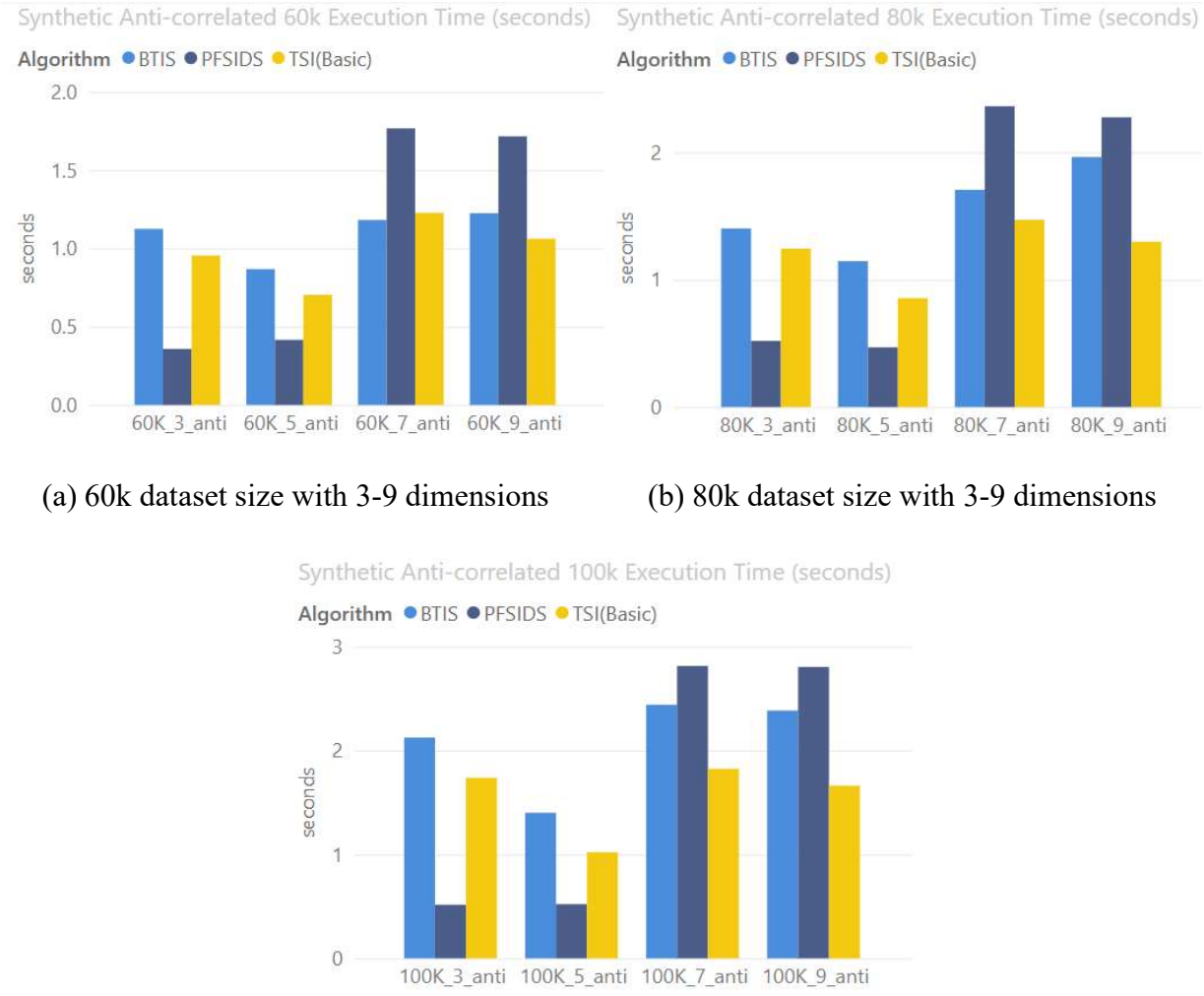
This chapter presents and discusses the results of each benchmark and compares the performance of each algorithm along execution time, memory usage and the number of comparisons. The variables of interest are the number of dimensions, the total size of the input, and the correlation between dimensions. The analysis section will discuss the strengths and weaknesses of each algorithm, and seek to explain why an algorithm may perform better or worse in a particular context.

4.2 Synthetic Datasets Results

Thirty-six different combinations of synthetic data were tested on each algorithm. For each type of data correlation (anti-correlated, correlated, random), four different dimensions (3, 5, 7 and 9) and three different input sizes (60KB, 80KB, 100KB) were used. It is clear from the results of these benchmarks, that correlation, input size and dimensionality impact the performance of each algorithm.

Figures 4.1, 4.2, and 4.3 concerning the results of the anti-correlated synthetic data, increases in dimensions had a large impact across all metrics. PFSIDS has the most significant jump in execution time, memory usage and number of comparisons in the 7 dimension and 9 dimension datasets, performing worse than BTIS and TSI. However, PFSIDS performed considerably better than TSI and BTIS in the 3 and 5 dimensional anti-correlated datasets across all metrics. Garbage collection allocation rate in the 7 and 9 dimensional datasets were lowest for

the BTIS algorithm and highest for PFSIDS. Execution time was the lowest in the 7 and 9 dimensional datasets for TSI. PFSIDS had a considerably lower number of comparisons for 3 and 5 dimensional data, but saw a drastic increase in the 7 and 9 dimensional datasets. TSI did slightly worse in 7 dimensional data compared to BTIS, but BTIS was slightly worse than TSI with 9 dimensional data.

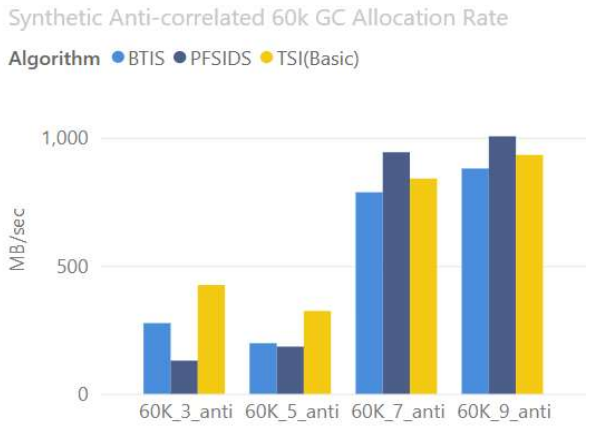


(a) 60k dataset size with 3-9 dimensions

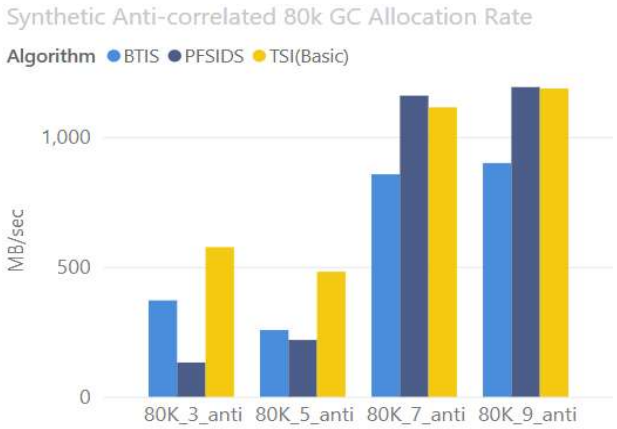
(b) 80k dataset size with 3-9 dimensions

(c) 100k dataset size with 3-9 dimensions

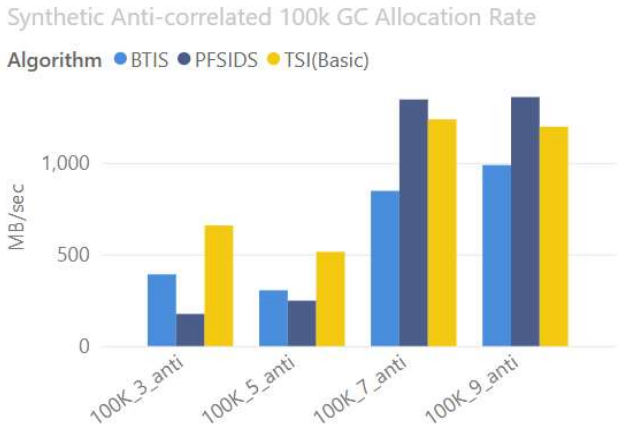
Figure 4.1 The Effect of Number of Dimensions on the Execution Times for Anti-correlated Synthetic Data



(a) 60k dataset size with 3-9 dimensions

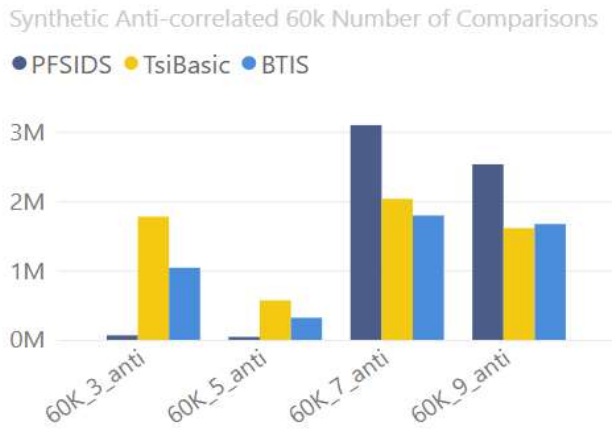


(b) 80k dataset size with 3-9 dimensions

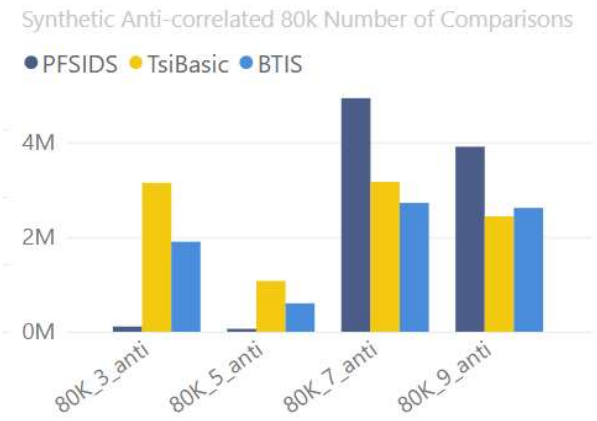


(c) 100k dataset size with 3-9 dimensions

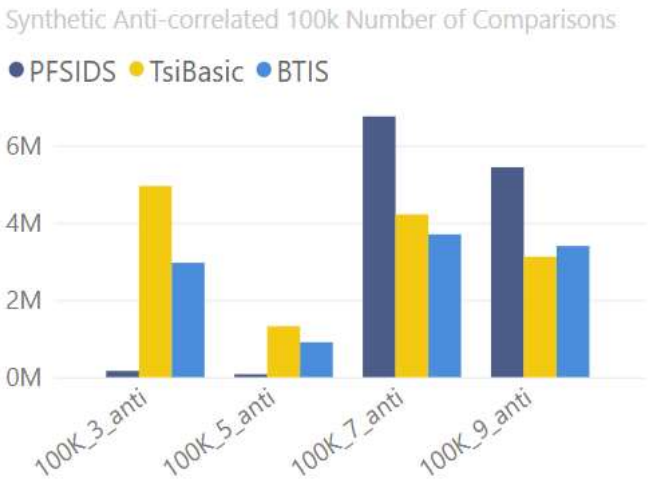
Figure 4.2: The Effect of Number of Dimensions on GC Allocation Rates for Anti-correlated Synthetic Data



(a) 60k dataset size with 3-9 dimensions



(b) 80k dataset size with 3-9 dimensions



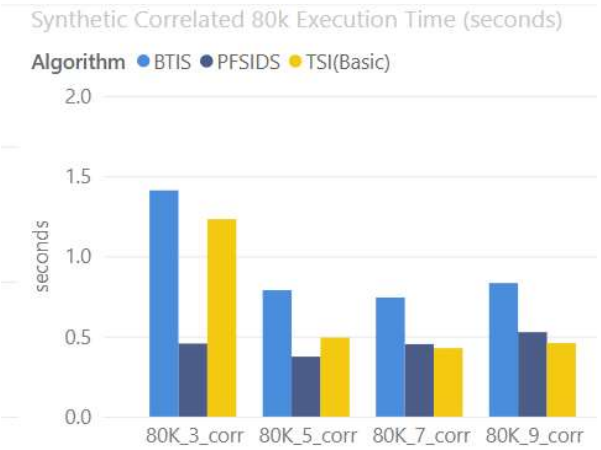
(c) 100k dataset size with 3-9 dimensions

Figure 4.3 The Effect of Number of Dimensions on Number of Comparisons for Anti-correlated Synthetic Data

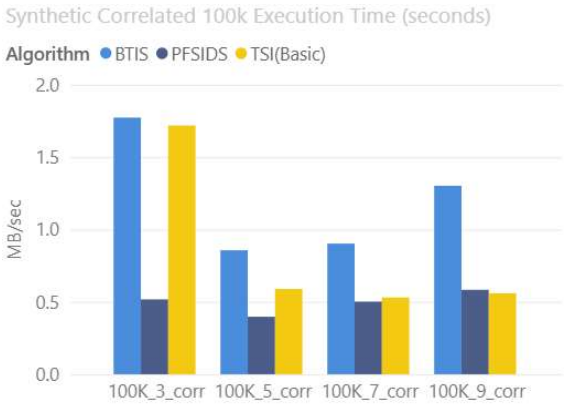
The experimental results presented in Figures 4.4, 4.5, and 4.6 show the outcomes for the correlated synthetic data. We can notice that the performance of PFSIDS remains stable as dimensions increase, while the performance of TSI and BTIS appear to improve. TSI outperforms BTIS in the 3 and 5 dimensional cases regarding execution time, but both were outperformed by PFSIDS. TSI and BTIS both see a significant decrease in execution time in the 7 and 9 dimensional

cases, with TSI slightly outperforming PFSIDS, and BTIS being the slowest of the three. When looking at garbage collector allocations, TSI and BTIS have the highest allocation rates in the 3 dimensional cases, have a sharp decline in the 5 dimensional case, and then steadily rise in the 7 and 9 dimensional case. PFSIDS sees a minor increase in memory allocations as the number of dimensions increases.

(a) 60k dataset size with 3-9 dimensions



(b) 80k dataset size wih 3-9 dimensions

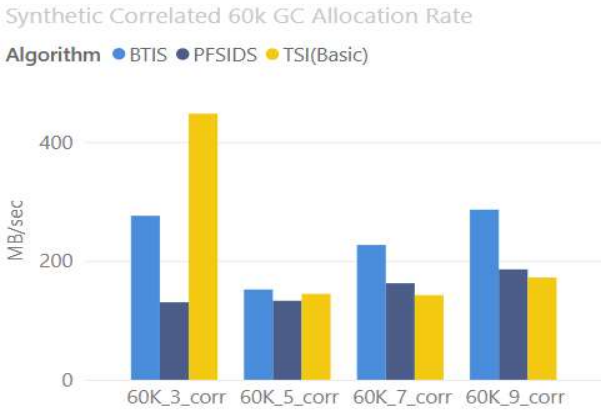


(c) 100k dataset size with 3-9 dimensions

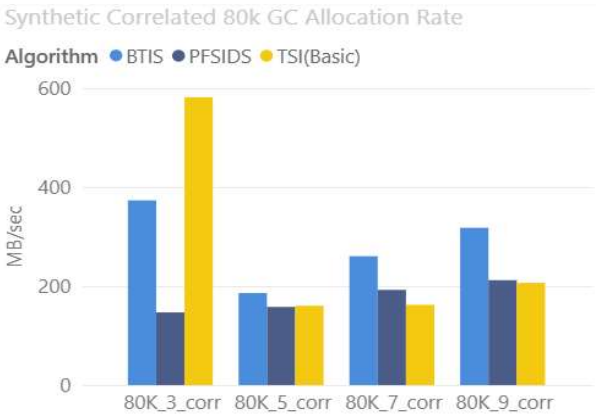
Figure 4.4: The Effect of Number of Dimensions on the Execution Time for Correlated Synthetic Data

As with execution time, TSI has slightly better memory allocation rates in the 7 and 9 dimensional case than PFSIDS, and BTIS performs worse than both across all cases. For the number of comparisons, TSI and BTIS have considerably more than PFSIDS in the 3 dimensional

case, but both have a drastic improvement in the 5, 7 and 9 dimensional cases. TSI only has slightly more comparisons than PFSIDS in the 5, 7 and 9 dimensional cases, and BTIS has noticeably more comparisons than PFSIDS and TSI.



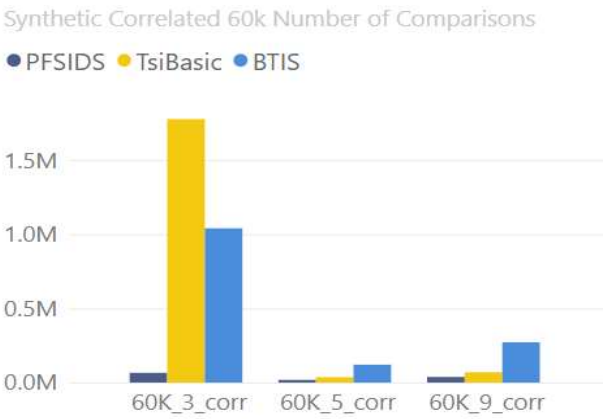
(a) 60k dataset size with 3-9 dimensions



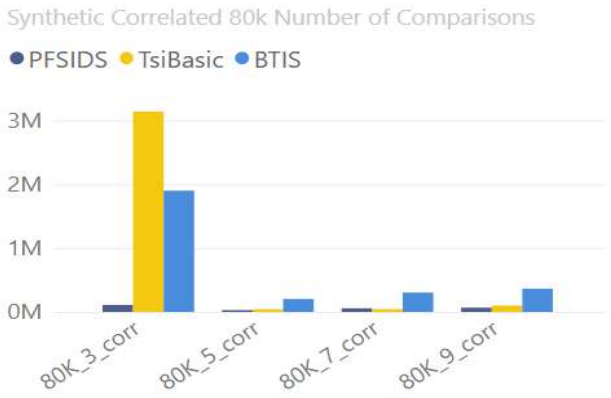
(b) 80k dataset size with 3-9 dimensions

(c) 100k dataset size with 3-9 dimensions

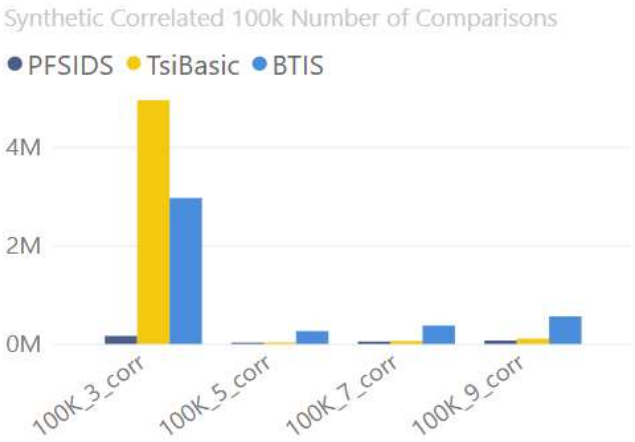
Figure 4.5: The Effect of Number of Dimensions on the GC Allocation Rates for Correlated Synthetic Data



(a) 60k dataset size with 3-9 dimensions



(b) 80k dataset size with 3-9 dimensions



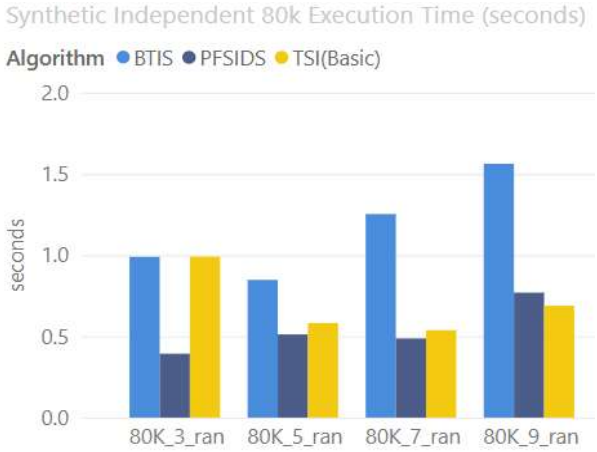
(c) 100k dataset size with 3-9 dimensions

Figure 4.6: The Effect of Number of Dimensions on the Number of Comparisons for Correlated Synthetic Data

Looking at the results of the synthetic random data in Figures 4.7, 4.8, and 4.9, we observe a similar pattern with correlated data. As dimensionality increases, PFSIDS has a stable increase in execution time, memory allocation and number of comparisons. TSI and BTIS perform considerably worse in the 3 dimensional case, see a drastic performance improvement in the 5 dimensional case, and then have a steady decrease in performance as dimensionality increases. BTIS performs worse across all cases in terms of execution time than TSI and PFSIDS, and TSI slightly outperforms PFSIDS in the 9 dimensional case. In regards to garbage collector allocations,

BTIS performed the worst in all cases, PFSIDS performed the best in the 3 dimensional case, and TSI performed the best in the 5, 7, and 9 dimensions case, allocating memory at a slightly lower rate than PFSIDS. For the number of comparisons, TSI has the most in the 3 dimensional case while PFSIDS has the lowest. TSI has fewer comparisons than PFSIDS in the 5 and 7 dimensional 100K dataset, and 5 dimensional 80K dataset. PFSIDS has fewer comparisons than BTIS and TSI in every other case.

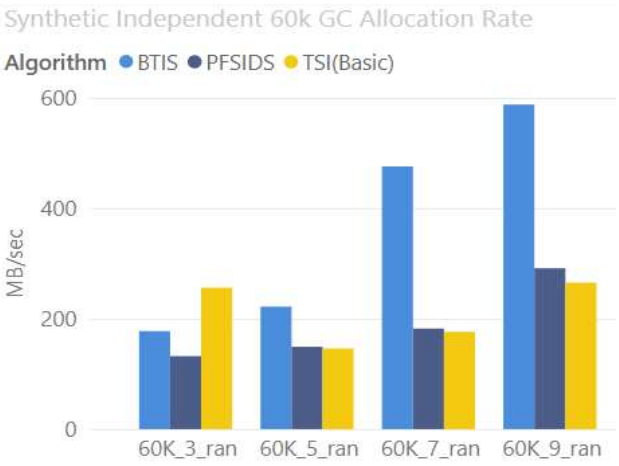
(a) 60k dataset size with 3-9 dimensions



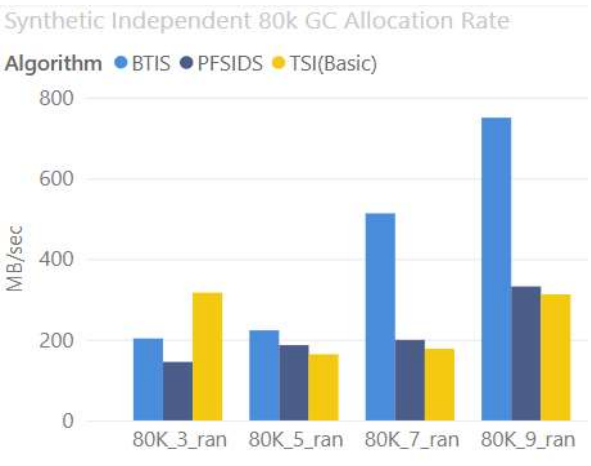
(b) 80k dataset size with 3-9 dimensions

(c) 100k dataset size with 3-9 dimensions

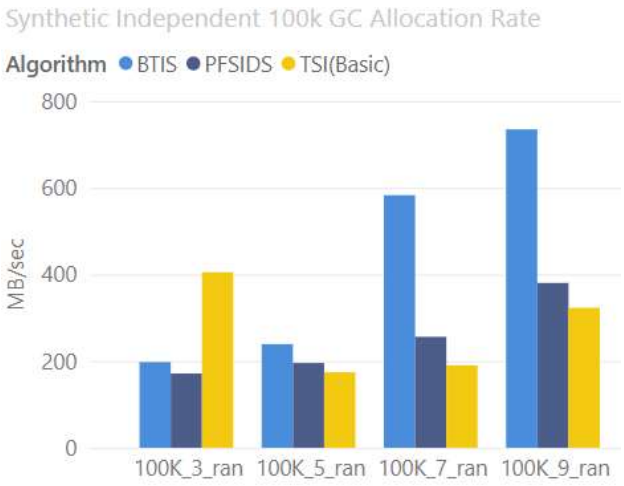
Figure 4.7: The Effect of the Number of Dimensions on the Execution Time for Independent Synthetic Data



(a) 60k dataset size with 3- 9 dimensions

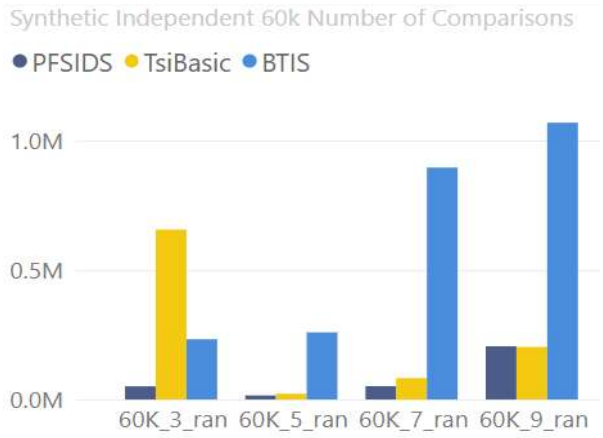


(b) 80k dataset size with 3-9 dimensions

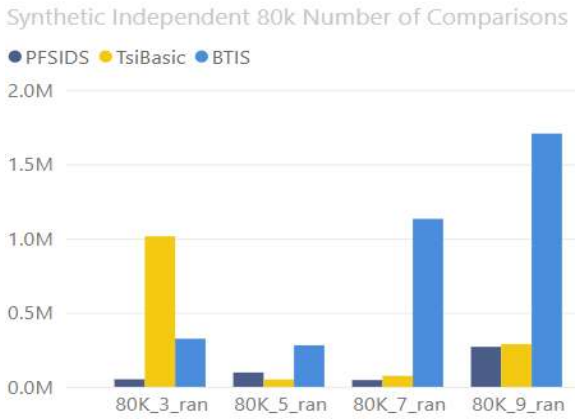


(c) 100k dataset size with 3-9 dimensions

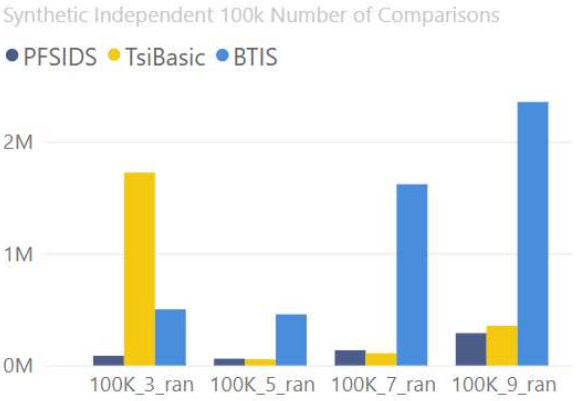
Figure 4.8: The Effect of Number of Dimensions son the GC Allocation Rate for Independent Data



(a) 60k dataset size with 3-9 dimensions



(b) 80k dataset size with 3-9 dimensions



(c) 100k dataset size with 3-9 dimensions

Figure 4.9: The Effect of the Number of Dimensions on Number of Comparison for Independent Synthetic Data

4.3 NBA Results

Referring to Figure 4.10, we can observe that the execution time of TSI and PFSIDS slightly increases as the number of dimensions increases. Conversely, BTIS exhibits a faster execution time as dimensions increase. BTIS has the quickest execution time in the 15 and 13 dimensional case, and PFSIDS has the fastest in the 11 dimensional case. TSI had the longest execution time in all cases. A similar trend is observed in the garbage collector allocation rates in Figure. 4.11, as the rate decreases for BTIS as dimensions increase, and slightly increases for TSI

and PFSIDS. TSI has a significantly higher memory allocation rate than both PFSIDS and BTIS in all cases. We observe the same trend again in the number of comparisons in Figure 4.12, however BTIS has more comparisons than PFSIDS in all cases, and TSI has significantly more comparisons than both algorithms in all cases.

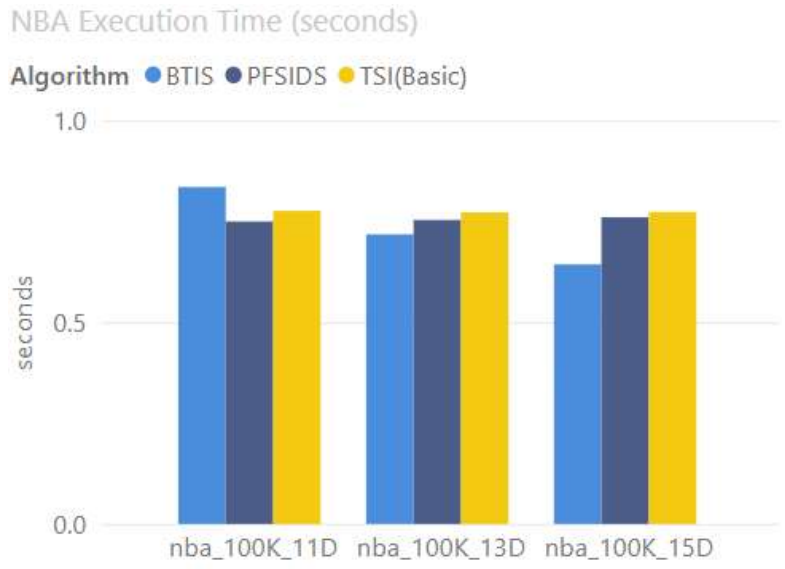


Figure 4.10: The Effect of Number of Dimensions on the Execution Time for NBA Real-world Data

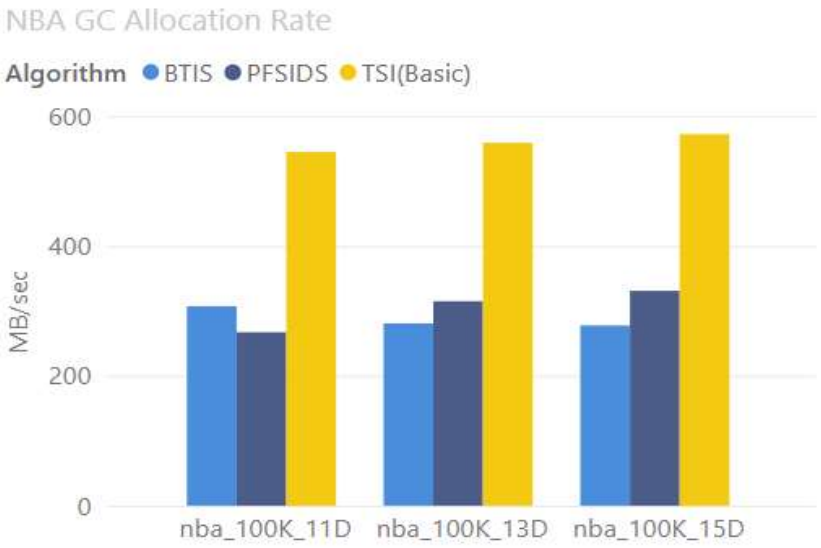


Figure 4.11: The Effect of the Number of Dimension on GC Allocation Rate for NBA Real-world Data

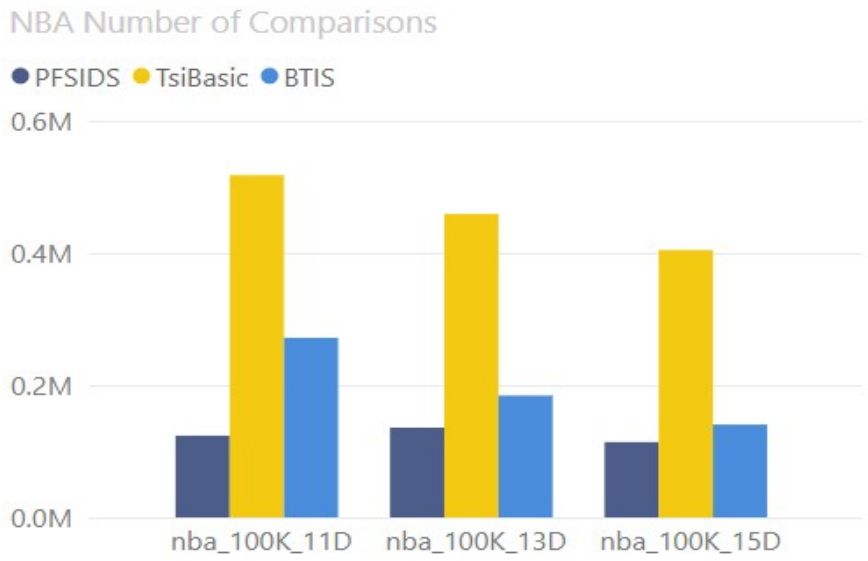


Figure 4.12: The Effect of Number of Dimension on Number of Comparison for NBA Real-world Data

4.4 COIL 2000 Results

From Figure 4.13, we can see that BTIS has the slowest execution times and TSI has the fastest, slightly faster than PFSIDS, in all cases. PFSIDS execution times slightly increase as dimensions increase, while TSI remains mostly the same. BTIS has variable changes in execution time, being slower in the 11 dimensional case, faster in the 13, and then slower again in the 15. Looking at Figure 4.14, we can see PFSIDS has the lowest allocation rates in the 11 and 13 dimensional case, TSI has the lowest in the 15 dimensional case, and BTIS has the highest allocation rate in all cases. BTIS has a significantly higher rate of memory allocation in all cases compared to TSI and PFSIDS. When comparing the number of comparisons in Figure. 4.5, we see BTIS has significantly more comparisons than TSI and PFSIDS, although the number of comparisons decreases as dimensions increase. PFSIDS has the fewest number of comparisons, and appears to have a stable amount of comparisons as dimensions increase.

Figure 4.13: The Effect of Number of Dimensions on the Execution Time for COIL 2000 Real-world Data

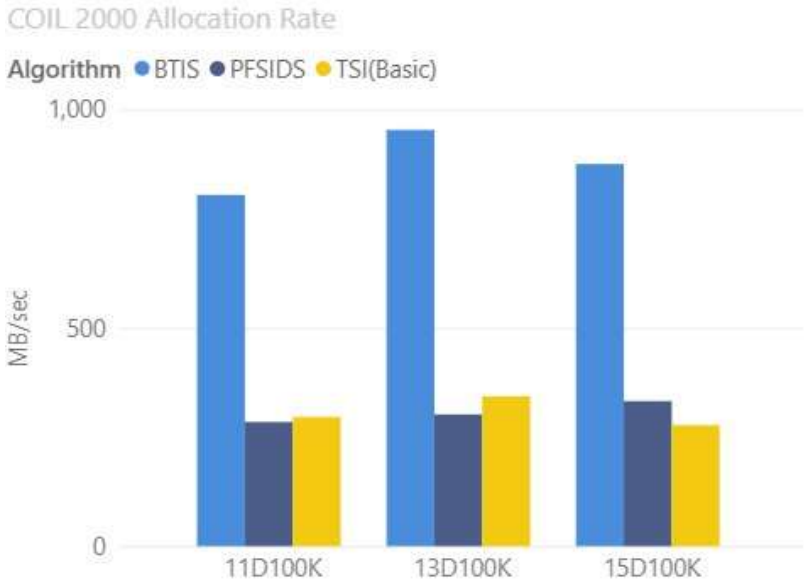


Figure 4.14: The Effect of the Number of Dimensions on the GC Allocation Rate for COIL 2000 Real-world Data

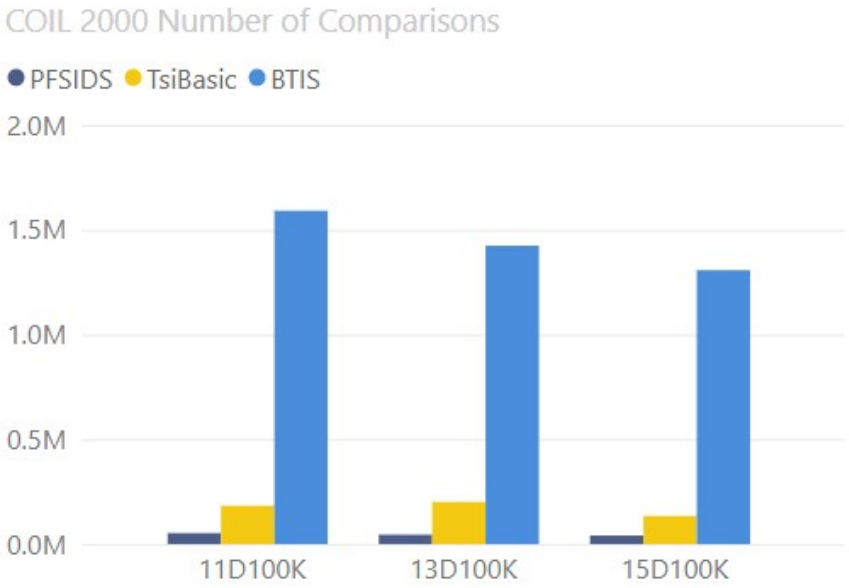


Figure 4.15: The Effect of the Number of Dimensions on Number of Comparison for COIL 2000 Real-world Data

4.5 Movielens Results

In Figure 4.16, the execution times of each algorithm are compared on the Movielens dataset. PFSIDS clearly performs the best at all input sizes by a large margin, taking at most 2 seconds on the largest data set (1200k), while TSI takes 40 seconds, and BTIS 26 seconds. This comes at the cost of the memory allocation rate as seen in Figure. 4.17 PFSIDS has a significantly higher rate of memory allocation that increases with the size of the dataset. TSI has the lowest memory allocation rate that appears to remain stable as the size of the dataset increases. BTIS has some variability in its memory allocation rate as dimensions change, but stays at a rate between TSI and PFSIDS. In the number of comparisons, as seen in Figure 4.18, PFSIDS clearly has the fewest number of comparisons for all sizes of the dataset, with BTIS having significantly more than both TSI and PFSIDS for all sizes.

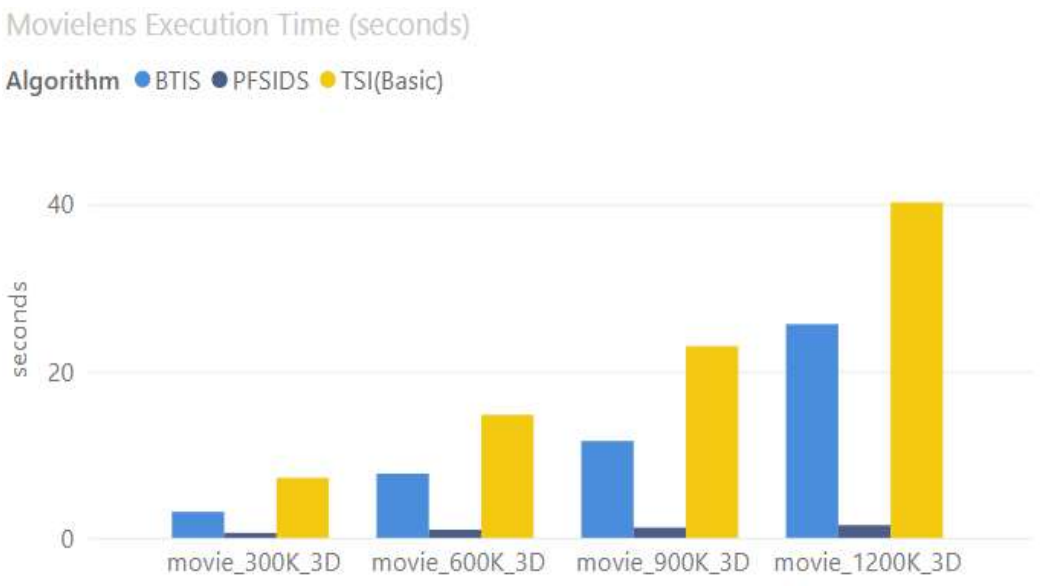


Figure 4.16: The Effect of the Dataset Size on Execution Time for Movielens Real-world Data

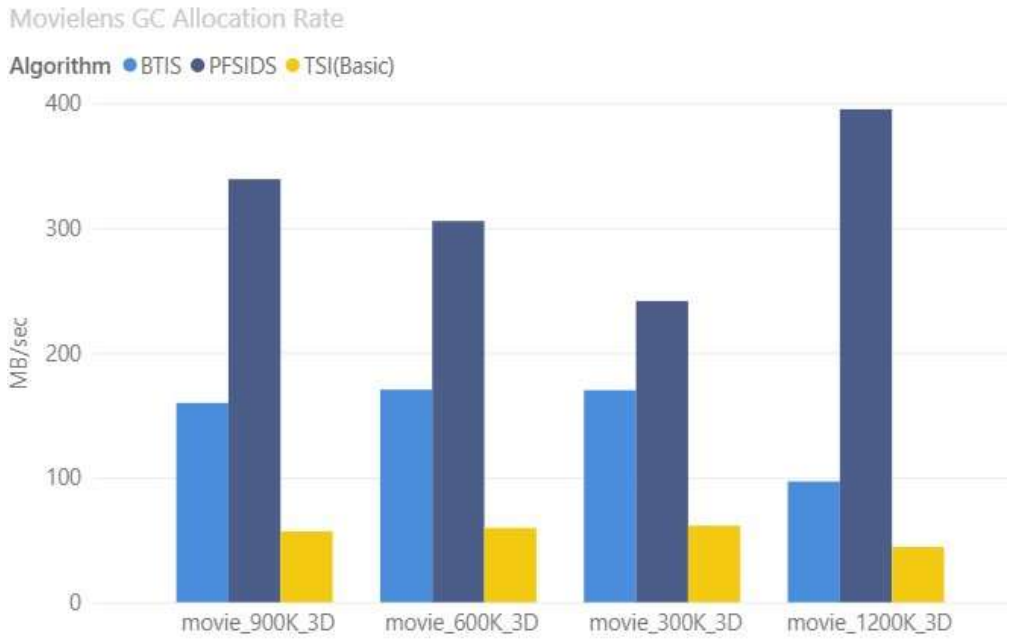


Figure 4.17: The Effect of the Dataset Size on the GC Allocation Rate for Movielens Real-world Data

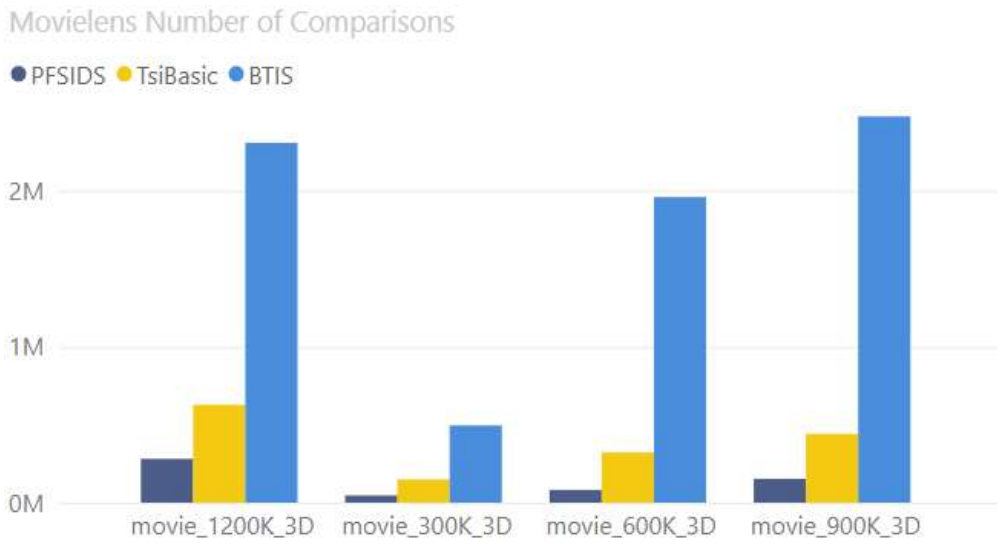


Figure 4.18: The Effect of the Dataset Size on Number of Comparison for Movielens Real-world Data

4.6 Analysis of Results

In most cases, PFSIDS outperforms BTIS and TSI in execution time and number of comparisons. It performed particularly poorly in the higher-dimensional, anti-correlated datasets however. This performance could be explained by the fact that when sorting anti-correlated points by dimension, a point having a high dominance capability with respect to one dimension, will also mean that it has lower dominance capability with respect to some other dimension. This could reduce the effectiveness of sorting as a point pruning strategy. Another important result is that PFSIDS had a much larger memory footprint than TSI and BTIS in the Movielens dataset, which were the largest datasets tested. This is not an unsurprising result, as PFSIDS requires the creation of an in-memory index that contains every point k times, with k being the number of dimensions of the data.

TSI performed the second best in general on the benchmarks, and did slightly better than PFSIDS in the higher dimensional synthetic datasets. However, the synthetic datasets are relatively

small inputs, and the asymptotic differences in runtime became much more pronounced in the larger Movielens dataset, where TSI performed much worse than both PFSIDS and BTIS. For smaller high dimensional datasets, TSI may be a better choice irrespective of how the data may be correlated. Another benefit of TSI over PFSIDS, unrelated to the benchmarks, is that it is a fairly straightforward algorithm to implement as it does not require building an index or some other auxiliary data structure like PFSIDS and BTIS. What was somewhat surprising was the memory allocation rate being higher than PFSIDS in some cases. TSI does not require any additional data structures, aside from a set data-structure that is iteratively built during the sequential scans. It's possible that the need to load the entire data set into memory twice in some scenarios results in more memory allocations than creating PFSID's index. Some of this variability could be due to the nature of the JVM's garbage collector and the single-shot benchmarks however. We should generally expect the least amount of memory usage from TSI, so this may indicate a need to tweak the parameters of the benchmarks.

BTIS generally performed the worse on the benchmarks, but performed noticeably better on the very high dimensional NBA dataset. The combination of high dimensionality, a low missing rate, and a high correlation among the dimensions may have provided good conditions for BTIS to succeed. A low missing rate may be important for BTIS to do well, as it may translate to fewer categories of data to classify. The classification and bucketing process introduces significant overhead, and as many as 2^d - 1 different buckets could exist. However, BTIS did not perform well in the COIL 2000 benchmark, a dataset that also has a relatively low missing rate and high dimensionality. An important aspect of the data, namely the number of different combinations of missing dimensions, probably plays a role in how well BTIS may perform. Another possible consideration is that there is a high initial overhead in the creation of the classification tree and

bucket data structures, whose creation is factored into the benchmarks execution time and memory allocation. In a scenario where such data structures were already prepared, the process of classification and computing the skyline itself may result in better results from BTIS.

CHAPTER FIVE

CONCLUSION

5.1 Introduction

In this chapter, we make recommendations as to when each algorithm is appropriate, and talk about possible future work. The results of our benchmarks give insights into the strengths and weaknesses of each algorithm and to differing approaches to dealing with incomplete data. We also explored the key differences in how each algorithm handles the problems that arise from computing the skyline on incomplete data. It is also clear that dimensionality, input size, and correlation impacts the performance of each algorithm. We also have several avenues for possible future work. We could use additional benchmarking metrics such as disk usage, give the JVM a warm-up period so that results are more reflective of real-world performance and explore the impact of missing rate on performance. Other ideas include investigating how the number of unique combinations of missing dimensions in the data impacts BTIS, test the more advanced pruning version of TSI, and benchmarking specific portions of each algorithm, such as the overhead of creating the classification tree in BTIS.

5.2 Conclusions

We recommend using PFSIDS for most cases, except when dealing with high dimensional, anti-correlated data. TSI is fine for smaller datasets and can give predictable performance irrespective of the data's correlation. BTIS unfortunately did not appear to have any clear pattern established for a context in which it excels in. Sorting appeared to be a better strategy for pruning tuples than BTIS's approach with bucketing and optimal virtual points. While in Yuan et. al, BTIS outperformed SIDs, another sorting-based skyline algorithm, the technique of ranking by

cumulative complete dimensions in PFSIDs appeared to have made a significant difference. More specifically, sorting not by just the values of each dimension, but also by cumulative complete dimensions was more effective than bucketing. TSI's approach for handling missing data was novel in that it's very straightforward and requires no additional data structures, but clearly had the worse performance on larger datasets. This is not unexpected given its O (n^2) time complexity. The pruning-based TSI algorithm, which processes pruning tuples with a high dominance capability from a min-heap, would have definitely fared better. However, it is important to notice the similarity between this version of TSI and PFSIDS, as using a min-heap is in itself a form of sorting. The key innovation of TSI is in how simply it handles the problems of intransitivity and cyclic dominance. BTIS clearly has the most initial overhead of the three algorithms. The need to model the classification tree on a sample of the dataset, as well as create buckets for each combination of missing values, likely caused it to perform poorly on most benchmarks. BTIS would probably perform better in a scenario where it could model the data once, and then each time the skyline needed to be computed, it could use the already prepared classification tree. This means that BTIS is probably best suited for static databases, as a dynamic environment would necessitate re-modelling the data every time a change was made.

5.3 Additional Metrics and JVM Warmup Time

Since a JVM language was used to benchmark these algorithms, single-shot benchmarks may not be the most accurate, as the JVM performs better on code paths that are "warmed up", or already run a few times prior. The cold execution times are still valid for the use of comparison however. Investigating I/O usage could be interesting as well, but would also require the use of much larger datasets to necessitate batch processing of the data. With the size of the data used in

the study, it did not make sense to look into I/O usage, as the entire datasets could easily fit into memory. While missing rates of the data were mentioned, they were not controlled for in the benchmarks. Exploring the performance impact of the missing rate could thus be a good direction for future work. A feature of the data not mentioned at all is the number of different combinations of missing dimensions. This should only affect the performance of BTIS, as it relies on classification of the data by its missing dimensions, but might impact the other algorithms nonetheless.

5.4 Considerations for BTIS and TSI

An aspect of BTIS that likely led to its poor performance was its extensive setup. Since BTIS had to remodel and rebuild its classification tree on each benchmark, the results may not be the most indicative of real-world performance. Separately benchmarking the tree classification portion and the actual skyline computation would give a better idea of BTIS's true performance. However, PFSIDS rebuilding its index on each run was also part of the benchmark, so the results still allow for fair comparison. Finally, testing the pruning version of the TSI algorithm that uses a min-heap to remove dominated tuples is obviously of interest. Its exclusion from this study was due to the interest of time and the complexity of implementing the algorithm.

References

Adhikari, Deepak, et al. “A comprehensive survey on imputation of missing data in internet of things.” *ACM Computing Surveys*, vol. 55, no. 7, 15 Dec. 2022, pp. 1–38, https://doi.org/10.1145/3533381.

Alwan, Ali, et al. “Missing Values Estimation for Skylines in Incomplete Database.” *The International Arab Journal of Information Technology*, vol. 15, 29 Nov. 2015, pp. 66–75.

Borzsony, S., et al. “The skyline operator.” *Proceedings 17th International Conference on Data Engineering*, 2001, https://doi.org/10.1109/icde.2001.914855.

Bharuka, Rahul, and Sreenivasa Kumar. “Finding Skylines for Incomplete Data.” *Australasian Database Conference*, 2013, pp. 109–117.

Choi, Wonik, et al. “Multi-criteria decision making with Skyline Computation.” *2012 IEEE 13th International Conference on Information Reuse & Integration (IRI)*, Aug. 2012, pp. 316–323, https://doi.org/10.1109/iri.2012.6303026.

Chomicki, J., P. Godfrey, et al. “Skyline with presorting.” *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*, Apr. 2003, pp. 717–719, https://doi.org/10.1109/icde.2003.1260846.

Chomicki, Jan, et al. “Skyline queries, front and back.” *ACM SIGMOD Record*, vol. 42, no. 3, 17 Oct. 2013, pp. 6–18, https://doi.org/10.1145/2536669.2536671.

Dehaki, Ghazaleh Babanejad, et al. “Efficient computation of skyline queries over a dynamic and incomplete database.” *IEEE Access*, vol. 8, 24 July 2020, pp. 141523–141546, https://doi.org/10.1109/access.2020.3011652.

Ding, Linlin, Shu Wang, et al. “Efficient K-dominant skyline query over incomplete data using mapreduce.” *Frontiers of Computer Science*, vol. 15, no. 4, 16 Apr. 2021, https://doi.org/10.1007/s11704-020-0122-x.

Ding, Linlin, Xiao Zhang, et al. “CROWDSJ: Skyline-join query processing of incomplete datasets with crowdsourcing.” *IEEE Access*, vol. 9, 25 Apr. 2021, pp. 73216–73229, https://doi.org/10.1109/access.2021.3079324.

Donald Kossmann, Frank Ramsak, et al. “Shooting stars in the sky: an online algorithm for skyline queries.” *In Proceedings of the 28th international conference on Very Large Data Bases (VLDB '02)*, 2002, pp. 275–286.

Emmanuel, Tlamelo, et al. *A Survey on Missing Data in Machine Learning*, 17 June 2021, https://doi.org/10.21203/rs.3.rs-535520/v1.

Fattah, H.M. Abdul, et al. "Weighted top-K dominating queries on highly incomplete data." *Information Systems*, vol. 107, 15 Feb. 2022, p. 102008, https://doi.org/10.1016/j.is.2022.102008.

Gulzar, Yonis, and Ali A. Alwan. "CIDS: An efficient algorithm for processing skyline queries for partially complete data in cloud environment." *IEEE Access*, vol. 10, 14 June 2022, pp. 66449–66466, https://doi.org/10.1109/access.2022.3185087.

Gulzar, Yonis, et al. "IDSA: An efficient algorithm for skyline queries computation on dynamic and incomplete data with changing states." *IEEE Access*, vol. 9, 12 Apr. 2021, pp. 57291–57310, https://doi.org/10.1109/access.2021.3072775.

He, Jingxuan, and Xixian Han. "Efficient skyline computation on massive incomplete data." *Data Science and Engineering*, vol. 7, no. 2, 3 Apr. 2022, pp. 102–119, https://doi.org/10.1007/s41019-022-00183-7.

Khalefa, Mohamed E., et al. "Skyline query processing for Incomplete Data." *2008 IEEE 24th International Conference on Data Engineering*, Apr. 2008, pp. 556–565, https://doi.org/10.1109/icde.2008.4497464.

Liu, Chuang-Ming, et al. "Priority-based skyline query processing for incomplete data." *25th International Database Engineering & Applications Symposium*, 14 July 2021, pp. 204–211, https://doi.org/10.1145/3472163.3472272.

Luo, Y., Lu, HX., Lin, X. (2004). A Scalable and I/O Optimal Skyline Processing Algorithm. In: Li, Q., Wang, G., Feng, L. (eds) Advances in Web-Age Information Management. WAIM 2004. Lecture Notes in Computer Science, vol 3129. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-27772-9_23

Miao, Xiaoye, et al. "Answering skyline queries over incomplete data with crowdsourcing." *IEEE Transactions on Knowledge and Data Engineering*, vol. 33, no. 4, 1 Apr. 2021, pp. 1360–1374, https://doi.org/10.1109/tkde.2019.2946798.

Papadias, Dimitris, et al. "An optimal and progressive algorithm for Skyline Queries." *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data,* 9 June 2003, pp. 467–478, https://doi.org/10.1145/872757.872814.

Rahman, Md. Sazedur, and K. M. Azharul Hasan. "Computing skyline query on Incomplete Data." *Lecture Notes in Networks and Systems*, Mar. 2024, pp. 657–672, https://doi.org/10.1007/978-981-99-8937-9_44.

Tiakas, Eleftherios, et al. “Skyline queries: An introduction.” *2015 6th International Conference on Information, Intelligence, Systems and Applications (IISA)*, July 2015, pp. 1–6, https://doi.org/10.1109/iisa.2015.7388053.

Yuan, Dengke, et al. “Skyline query under multidimensional incomplete data based on Classification Tree.” *Journal of Big Data*, vol. 11, no. 1, 12 May 2024, https://doi.org/10.1186/s40537-024-00923-8.