Rheinland-Pfälzische
Technische Universität
Kaiserslautern
Landau

Query Processing and Cardinality Estimation over Data Summaries

PhD Thesis

Thesis approved by
the Department of Computer Science
University of Kaiserslautern-Landau
for the award of Doctoral Degree
Doctor of Engineering (Dr.-Ing.)

to

Damjan Gjurovski

Date of Defense	14.05.2025
Dean	Prof. Dr. Christoph Garth
Reviewer	Prof. Dr.-Ing. Sebastian Michel
Reviewer	Junior Prof. Dr. Sandra Geisler

DE-386

Abstract

In this thesis, we propose methods and optimizations for efficient query processing and cardinality estimation over data summaries. For this reason, we put forth algorithms that work on a substantial amount of static data and methods tailored for streaming environments, where the data is constantly being generated and arrives over time. When working in streaming environments, we first address the problem of natural join computation over streams of semi-structured documents. We initially propose an efficient and scalable partitioning algorithm that uses the main principles of association analysis to identify co-occurring attribute-value pairs within the documents. To compute the join pairs, we put forth a novel natural join algorithm that comes in two flavors. The join algorithm utilizes a frequent pattern tree (FP-tree) to store a compact representation of the schema-free documents. Differently, we address the problem of data stream summarization on edge devices to reduce network traffic and speed up further processing. Instead of emitting the whole data stream, our SoftSieving approach computes item-based summaries using core items. Core items of a data stream are the items with the highest values for a given monotone submodular utility function. When working with vast static data, we address approximate query processing and cardinality estimation. We propose a versatile approach to lightweight, approximate query processing by learning compact but tunably precise representations of larger quantities of tuples coined bubbles. Bubbles are tunable regarding the compactness of enclosed tuples as well as the granularity of statistics and the way they are instantiated. Our solution models the content of the bubbles using Bayesian networks and deep autoregressive models. We have also identified an optimization for deep autoregressive models for the task of cardinality estimation. We overcome the limitations of such models when handling queries involving range predicates by circumventing the typically applied, time-consuming estimation algorithm. Consequently, we present our hybrid approach, Grid-AR, that fuses deep autoregressive models with a grid-based structure.

Contents

Chapter 1

Introduction

In the current data-driven world, organizations increasingly operate in environments where vast amounts of information are generated and processed both locally and in real-time streaming contexts. Consider a large retail company with different entities and hundreds of physical stores across many countries. Every store would be equipped with IoT devices like smart shelves, inventory sensors, and in-store cameras. Together with information that comes from user purchases and internal management, this modern company would be processing additional data from the stores. Considering this setting, we have identified three distinct opportunities from which such a company can benefit.

In such modern organizations that span multiple departments, data will be generated and stored in various formats. These formats can range from structured relational databases to semi-structured JSON files and even unstructured textual documents, such as logs or reports. Combining this diverse data is essential for obtaining a unified understanding of the business and uncovering insights for strategic decisions. When dealing with schema-free documents, joining data from different departments and systems presents unique challenges. Such documents often have varying structures, inconsistent fields, and different data representations based on individual systems, processes, and regional requirements. When considering logs, with detailed analysis, the company can gain real-time insight into potential issues and develop long-term solutions. For example, by analyzing IT support logs that contain information about unresolved software issues, the company can gain an in-depth understanding of common problems across regions. This can initially trigger an immediate action, such as dispatching a unit to resolve the problem. In the long term, the company can develop a standardized approach and facilitate future handling of such issues. To this end, we propose algorithms for *joining schema-free data in streaming environments*. First, we put forth a novel partitioning algorithm to organize data with the end goal of parallel computation of natural joins. Additionally, we develop two alternatives of a local join algorithm that produces the join results fast while storing minimal information.

Making use of the numerous store-specific information can bring direct monetary benefits for the company. Analyzing the data from in-store cameras, smart shelves, or other IoT sensors can deliver insights regarding promotions, inventory management, and security monitoring for the stores. Such benefits can only be valuable for the store if they are obtained in real time during uninterrupted operations. Thus, by leveraging computations directly on the edge devices, the company can gain speed and cost efficiency. This improved responsiveness is due to minimizing or avoiding constant data transfer to centralized locations while speeding up subsequent processing. To address this, we propose an algorithm for *data summarization that works directly on edge devices* by considering and addressing the limitations of the existing related work [12, 23, 84]. Our SoftSieving approach computes item-based summaries using core items.

The individual stores will generate vast amounts of data from customer interactions and sales transactions. In specific scenarios, processing substantial amounts of data may not be imperative. Instead, approximate answers would be enough to attain valuable insights quickly and drive business decisions. For example, to better promote products, the store managers might be interested in analyzing the top-selling products across multiple stores or calculating the average sales volume for a specific product category and region. The retail company can adopt approximate query processing techniques to address these challenges. Instead of processing every transaction in detail, approximate query processing enables the company to obtain reliable insights in a fraction of the time compared to traditional data processing methods. For this purpose, in this thesis, we address *approximate query processing for queries over databases consisting of multiple relations*. We introduce the concept of *bubbles* that represent groups of tuples from the tables. Instead of storing the exact data, we summarize the information of every bubble and use the respective summaries for approximate query processing. For summarization, we employ Bayesian networks and deep autoregressive models. In addition, explicitly focusing on cardinality estimates, we propose a hybrid approach that combines deep autoregressive models with a multidimensional grid structure. Our approach, termed Grid-AR, separates the handling of columns typically used in range predicates from the ones usually used in equality predicates. This way of operation circumvents the iterative estimation algorithms used in deep autoregressive models.

1.1 Problem Statement

In this thesis, we address query processing in terms of joining of data in streaming environments and data summarization in resource-constrained streaming environments. We further explore approaches for answering aggregate queries and cardinality estimation over summaries of static data. The abovementioned problems are addressed independently and are more formally defined as follows.

In the streaming environment, as the working model we assume an incoming stream S of documents d_i and a set of m machines over which the computation is to be distributed. Each document from the incoming stream of documents $d_i \in S$ consists of an unordered set of attribute-value pairs $d_i = \{a_1 : v_1, a_2 : v_2, \ldots\}$. Concerning our work on *joining of data streams*, same as for natural inner joins in traditional RDBMS, two documents are part of the join result, if and only if they have identical values for the attributes (columns) they share. In this work, documents that do not share any attributes are excluded from the join result, in contrast to the behavior in ANSI SQL where the cross product is computed in this case. To compute the joins, we consider **window joins**, where two documents can only be joined if they belong to the same time (or count-based) window. In particular, we make use of tumbling windows where the data for every window is not overlapping with the data from the previous or successive window. More formally, with $att(d_i)$ denoting the attributes of document d_i, and $d_i(a)$ referring to the value of attribute $a \in att(d_i)$, a pair (d_i, d_j) is a result of the natural join if the following two conditions hold:

$$att(d_i) \cap att(d_j) \neq \emptyset \quad and \tag{1.1}$$

$$\forall a \in att(d_i) \cap att(d_j) : d_i(a) = d_j(a) \tag{1.2}$$

We realize the task of join computation by splitting it into two parts. First, we **partition the data** to group potentially joinable documents on the same machines. The partitioning of the data allows the workload to be spread out, improving scalability and enabling parallel computation of the incoming stream. **The task of data partitioning** demands forming partitions that act as guideposts to lead incoming documents to the machine responsible for a partition. We form m partitions, one per machine. Partitions are expressed as sets of attribute-value pairs, too[1]. Depending on how the partitions are created, some documents might need to be sent to multiple machines, to ensure correct join results. We call this **replication** and aim at keeping it low. Additionally, we aim at fair **load balancing** across the compute nodes, which can be sometimes in conflict with low (or zero) replication. But one without the other goal can lead to impractical situations, like reaching zero replication by assigning all documents to a single machine or achieving perfect load balancing by sending all documents to all machines, to highlight the extreme cases. To produce the actual join result, we perform the second task of **local join computation** within every machine. The task of local join computation needs to ensure that documents that arrived at one compute node are correctly and efficiently matched to documents that arrived earlier, then need to be stored (and indexed) to be matched to incoming documents next. Since we consider stream joins, a feasible approach has to adapt to changing data characteristics. That is, appearance of previously unseen attribute-value combinations or different co-occurrences between attribute-value pairs which have a negative impact on the partition quality (in terms of load balancing and replication). The case of a join between two different local datasets is defined analogously. When presenting our solution, we do not treat the algorithms separately for the two cases but discuss how joins involving two or more relations can be optimized in Section 4.5.3.

When working in a resource restricted streaming environment, where the data is generated at various edge devices, if there is a constant transfer of data, this will directly impact the time of the approach. To avoid transferring every item from $d_i \in S$ through the network, we aim at **computing item-based summaries** directly at the edge devices. The summaries will be computed using so-called core items, that are representative items that appropriately identify groups of items given a function f. More precisely, the core items will be computed as the items that have the highest value for a given monotone submodular utility function f. As utility functions, we will use the k-medoids and active sets. The task of computing the core items will be done in parallel on the $m-1$ edge devices such that one edge device will be responsible for consolidating and generating the global core item set. The computation of the core items will be performed over windows where we will consider tumbling windows. The aim is to deliver summaries of good quality by considering the utility value while reducing the time required to compute them.

When working with vast static data, we focused on providing quality estimates fast while maintaining less data for the tasks of **approximate query processing** and **cardinality estimation**. Unlike our previous work in the field of

[1]Note that, in fact, it would also be possible to express partitions solely by attributes and not by attribute-value pairs. However, the restriction to attributes would not exhibit enough ways to (evenly) distribute computation to nodes, at least not for data or application cases we have observed.

stream processing, we consider the whole dataset during creation and estimation and not windows of it. To address the task of *approximate query processing* in a static environment, we consider a database consisting of relations R_i, each having a set of attributes $A_1, \ldots, A_n$. Instead of directly working with the relations, we consider groups of tuples. We accomplish this by organizing tuples into so-called bubbles. To produce the estimates, we summarize the information for every bubble. Thus, to accomplish this, we learn for each bubble a concise but powerful enough representation that is used to answer *aggregation queries* in an approximate manner. In particular, we focus on the aggregation functions `SUM`, `AVG`, `MIN`, `MAX`, and `COUNT`. Queries may involve an arbitrary number of *equality joins*, i.e., foreign-primary key joins, and *equality or range predicates.* The goal is then to provide estimates of good quality by considering metrics such as the $q{-}error$, reduce the time needed for producing an estimate by measuring the time required to parse and answer the query, and lower the memory consumption on disk.

For the task of **cardinality estimation**, we consider individual relations which as previously, consist of an arbitrary number of numerical and textual attributes $R_i = [A_1, \ldots, A_n]$. We aim at answering `COUNT` queries over the individual relations. Thus, let $Q(R_i)$ be a query involving conjunction of predicates over any column of relation R_i, i.e., $Q(R_i) = \bigwedge_i R.A_i \,\theta\, v$, where $\theta \in \{=, >, <, \geq, \leq\}$ and v is a value usually in the domain of A_i. Let $|Q(R_i)|$ denote the cardinality, i.e., number of tuples qualifying for the query. The objective is to attain an accurate estimate of $|Q(R_i)|$, considering metrics such as the $q - error$ while reducing the time needed for producing estimates and the memory consumption.

1.2 Contributions and Publications

With this work, we make the following contributions to the areas of partitioning, joining, and data summarization in streaming environments and approximate query processing and cardinality estimation for static data:

- We introduce a novel, association-groups–based partitioning algorithm for handling schema-free data with the end goal of parallel computation of natural joins.
- We propose and thoroughly investigate two approaches for performing natural joins over the local data per machine to produce the join results fast while minimizing the memory overhead. The approaches utilize a modified frequent pattern (FP) tree as an underlying indexing structure that stores the content of the schema-free documents.
- We present our data summarization approach, SoftSieving, which operates on data streams in resource-constrained environments. By summarizing the streams through the computation of core items, the proposed approach drastically reduces the network traffic and speeds up further processing.
- We introduce a data summarization and estimation approach for handling approximate query processing and cardinality estimation when working with static data of substantial size. The proposed approach summarizes

groups of tuples using Bayesian networks or deep autoregressive models and makes use of these summaries for computing the result estimates.

- We propose Grid-AR, a novel hybrid approach for cardinality estimation in a static environment. The presented approach integrates a grid structure with a deep autoregressive model for separate handling of attributes typically used in range predicates from the ones typically used in equality predicates. Using the hybrid structure, we further present an algorithm that efficiently tackles the vaguely explored topic of cardinality estimation for range join queries.

The content of this thesis has been published on the following peer-reviewed conferences and workshops.

- We introduced our work on realizing natural joins in streaming environments detailing our partitioning and joining approach at the IEEE International Conference on Data Engineering (ICDE) in 2020 [60].
- We performed a detailed evaluation of the local join algorithm, introducing a new variant and expending the algorithms beyond self joins at the workshop on Scalable Cloud Data Management that was co-located with the IEEE International Conference on Big Data in 2021 [61].
- We published and presented our SoftSieving algorithm for data summarization when working with streams in resource-constrained environments at the European Conference on Advances in Databases and Information Systems (ADBIS) in 2022 [59].
- We motivated and presented our work for dealing with vast static data in the context of approximate query processing and cardinality estimation at the workshop on Exploiting Artificial Intelligence Techniques for Data Management that was co-located with the ACM International Conference on Management of Data (SIGMOD) in 2023 [62].

1.3 Outline

The remainder of this thesis is organized as follows. In Chapter 2, we detail the essential concepts and technologies that are necessary to understand this thesis. Chapter 3 describes the relevant related work.

Chapter 4 discusses our work on natural joins in a streaming environment. We initially introduce our partitioning algorithm that groups potentially joinable documents on the same machines. We then present the two different join algorithms performed in parallel at the individual machines. We further perform a detailed experimental evaluation showcasing the benefits of our proposed approaches in terms of load balance, execution time, and memory consumption.

In Chapter 5, we detail our approach for data stream estimation at edge devices. We initially motivate our solution by outlining the limitations of the existing related work when working in such environments. We then present the architecture of our proposed solution and perform an elaborate experimental evaluation with the state-of-the-art competitors.

Chapter 6 presents our solution for approximate query processing over static data. We initially explain how to organize the data into so-called bubbles before presenting solutions based on Bayesian networks and deep autoregressive models for summarizing the data in the individual bubbles.

In Chapter 7, we address the limitations of autoregressive models for handling numerical attributes typically involved in ranged query predicates. We then elaborate on our proposed solution that combines a grid structure with an autoregressive model while eventually showcasing the reduced memory consumption and improved estimation time. In addition, using the grid structure, we present an algorithm for estimating range join queries.

Finally, in Chapter 8, we conclude our work and discuss the potential future directions.

Chapter 2

Background

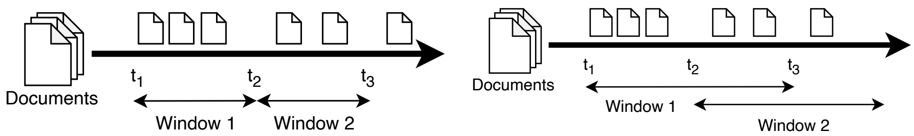
In the following, we will outline the background knowledge that is important for understanding the contents of this thesis. We will start by describing the stream processing model, covering window-based processing and the framework Apache Storm. We will then elaborate on frequent pattern trees and the textbook algorithms for their traversal. Furthermore, we will provide the necessary details for understanding stream summarization through the use of utility functions, especially focusing on submodular functions. Additionally, we will introduce and explain methods used in approximate query processing and cardinality estimation for dealing with vast data. In particular, we will detail Bayesian networks and deep autoregressive models as the building blocks for our proposed approaches.

2.1 Data Streams

2.1.1 Data Stream Processing

A data stream represents an unbounded flow of items that is arriving over time. Because of the high volume of data, it is impossible to store it and process it using conventional data processing algorithms. Since the data needs to be processed in near real-time, there is a shift from the traditional query processing native to relational database management systems. Instead of having single time queries, in stream processing, the queries are continuous and the data is constantly moving [55]. A continuous query is a standing query registered on a data stream that continuously produces results as new data arrives, according to a defined temporal scope. To accomplish this, the queries are executed over a snapshot of the data, or so called *windows*. For example, when working with temperature that is measured from various sensors, we might be interested in *continuously* emitting the average temperature every hour.

There exist different types of windows. The most prominent ones are *sliding* and *tumbling* windows. The data within a window can be grouped based on time of arrival, i.e., the last five minutes, or based on count, i.e., the last thousand tuples. A tumbling window groups the tuples from the stream based on a



(a) Tumbling window. (b) Sliding window.

Figure 2.1: Example for data stream windows. On the left-hand side is a tumbling window and on the right is a sliding window.

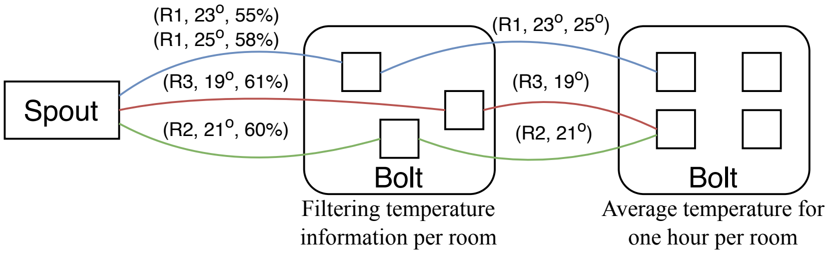


Figure 2.2: Example Apache Storm topology for computing the average temperature for a room including information about parallelism.

specific time or count such that there is no overlap between the tuples of different windows (Figure 2.1a). A sliding window is characterized by the window moving over the data from the stream based on a specified interval. In sliding windows, there is an overlap between the tuples of different windows (Figure 2.1b).

Data stream processing is especially suited for scenarios where approximate answers are enough since, typically, the input data is seen only once and not processed in multiple passes like in traditional database management systems. Hence, the focus is on devising algorithms that operate in real time and avoid the storage of the complete, potentially unbound, dataset [55].

2.1.2 Apache Storm

Apache Storm [2] is a distributed stream processing platform running parallel computations across a cluster of machines. Storm guarantees fault tolerance, guaranteed message delivery, and exactly-once tuple processing semantics. Applications realized in Apache Storm consist of components that are grouped into so-called Storm topologies. A storm topology can have two types of components, *Spouts* and *Bolts*, that work together to execute the logic of the application. The spouts represent the starting point of the topology, they are the source of the stream in the topology. The bolts process one or more streams and optionally emit new streams from their computations. The connection between the spouts and bolts is realized by using *stream groupings*.

An example Storm topology is depicted in Figure 2.2. Let us imagine that we want to compute the average temperature for the rooms in one building for every hour of the day. The spout reads sensor information from sensors placed within the rooms which consist of information about the temperature and the humidity. From the sensor readings, the spout creates tuples by adding the room name and propagates these tuples to the next component. In the filter bolt, since only the temperature is of interest, the information about the humidity is ignored, and based on the room name, the temperature information is forwarded to the bolt responsible for emitting the average temperature for every hour. This bolt, stores as state a list of the temperatures received in the last hour by implementing a tumbling window. At the end of the hour, the bolt emits the average temperature per room.

The parallelism in Storm is realized by specifying the number of instances for every component, i.e., tasks. This number is used by Storm for creating the required number of threads for every component. To perform the processing in parallel, the threads are split across the cluster of machines. From the depicted example in Figure 2.2, the small squares within the bolts represent tasks such that for the first bolt, there are three, and for the second, there are four tasks. For the spout there is only one task in the depicted example. The storm cluster which represents the distributed machines that execute the storm topologies consists of two node types. A central node, called nimbus, is responsible for distributing the data, assigning tasks to other nodes, and monitoring the nodes for failures. Supervisor nodes receive commands from the nimbus and direct them to worker processes. The worker processes execute respective parts of the topology. It is important to note that because Apache Storm is stateless, the communication between the nimbus and supervisors as well as the maintenance of the state is carried out by a separate framework called Apache ZooKeeper [1]. With the help of the ZooKeeper, nodes can be easily restarted after failure without affecting the topology.

The data between the components in the topology is transferred in the form of tuples. Tuples represent a simple list of named values. For receiving data, bolts need to register to the output stream of the appropriate spout or bolt. Storm provides different types of *stream groupings* that define how the tuples will be received by the multiple instances of the bolt.

- **Shuffle grouping**: randomly distribute tuples across the instances of the bolt such that every instance will receive an equal number of tuples.
- **Fields grouping**: partition the stream based on one or more fields specified in the grouping.
- **All grouping**: the stream is replicated across all of the instances of the bolt.
- **Direct grouping**: the producer of the tuples decides which instance of the subscriber will receive the tuple based on a unique identifier.

For the example in Figure 2.2, the spout is connected to the filtering bolt using shuffle grouping so it can happen that tuples about the same room end up on different tasks. The filtering bolt and the bolt computing the average temperature are connected using fields grouping on the first field of the tuple, i.e., the room name. Consequently, the measurements of one room will always end up on the same task.

For our work in resource restricted streaming environments, we will consider a fork on top of Apache Storm tailored for work on edge devices [52]. The reason for this is that Apache Storm can require too many resources that might not be available in typical edge devices. This adaptation introduces several changes that are more suitable for edge applications. In Apache Storm, each operation is assigned to a thread and scheduling is handled by the operating system, which is not aware of congestion inside the topology. When working on the edge, this can lead to high latency and backpressure. In EdgeWise [52], there is a congestion-aware scheduler assigning the operations to threads from a fixed-size worker pool. With that queue lengths are balanced and backpressure is minimized.

2.2 Frequent Pattern Trees

Association analysis focuses on identifying interesting relationships that are hidden within the data of interest. It is beneficial in various domains, including but not limited to the fields of analysis of purchase behavior, web mining, medical data analysis, and scientific data analysis. Association analysis is performed with the use of association rules that are implications of the form $X \rightarrow Y$, where X and Y are disjoint itemsets, i.e., $X \cap Y = \emptyset$ [152]. An itemset represents a set of items present in a transaction, such as, for example, all the products a user has purchased. To determine if a set of items is frequent, association analysis relies on the metric *support* that is used for analyzing association rules. More specifically, the support represents the number of transactions in which the items of the itemset appear. An itemset is **frequent** if its support is above a prespecified threshold.

Frequent Pattern Trees (FP-trees), first introduced by Han et al. [70], offer an innovative method for storing essential information about frequent patterns in a compact manner. Using FP-trees enables more efficient and faster association mining than using traditional techniques. Most existing mining methods are based on an Apriori-like approach, which operates on the principle that if a pattern of length k is not frequent in the database, then any super-pattern of length $k + 1$ will also never be frequent. Apriori-based approaches encounter limitations in cases involving numerous or long frequent patterns. These limitations arise due to the large number of candidate sets that must be generated for frequent itemsets and the repeated scans of the database. FP-trees, along with the corresponding FP-growth algorithm, address these inefficiencies by overcoming the drawbacks associated with Apriori-based methods.

To build an FP-tree, only two database scans are needed. The first scan is required to cover the items of all transactions and to calculate their support. As a successive step, the items are ordered by the support in descending order such that items that have support that is less than the prespecified threshold are removed. In the second scan, the frequent items of the transactions are added as branches of the FP-tree by following the fixed ordering of the items done in the preceding scan. Together with the FP-tree, a so-called header table is built. This table gives pointers for faster traversal of the nodes of the FP-tree that have equal labels. Following the algorithm for creating an FP-tree, it is evident that by storing the frequent items in an FP-tree, the need for repeated scans of the database is overcome. Furthermore, since the items are inserted as branches of the tree following the same strict ordering, the need to create separate branches for individual transactions is eliminated. This is the case since frequent items that are shared between transactions can be represented by the same nodes. For every node in the FP-tree, together with the label two more information will be kept. A *counter* indicating the number of transactions that use that node and a node link that references the next equally labeled node of the tree or *null* if there is no such node.

An example FP-tree and the associated header table are depicted in Figure 2.3. For the given example, it is assumed that the strict order of the items is $b \rightarrow a \rightarrow c$. At the beginning, the FP-tree is empty and consists of only the root node. Let us assume that the tree will be built for three transactions such that

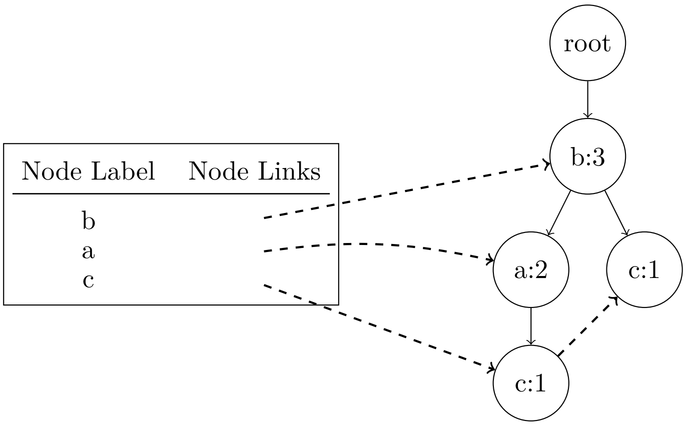


Figure 2.3: Example FP-tree.

transaction t_1 consists of items a, b, c, transaction t_2 consists of items a and b, and transaction t_3 consists of items b and c. When inserting transaction t_1 since b is the first item in the defined order, a new branch will be created having b as the direct child of the root, then node a child of node b which in turn will have one child labeled with c. The count for every node of this branch will be set to one indicating that only one transaction uses the nodes of this branch. In addition, since all these nodes do not exist in the header table, they will be inserted there. When working with the items from transaction t_2, the first node that will be examined is node b. Since the root already has an equally labeled node, a new node will not be created. Instead the existing node labeled with b will be reused. This is reflected by increasing the counter of the node to two. The same procedure is repeated for the next item of t_2 since node b has a child node labeled with a. For transaction t_3, since the node b does not have a child labeled with c, a new node will be created. In addition, the link of the existing node labeled with c will be updated to point to the newly created node.

Once the FP-tree is built, one can employ the FP-growth algorithm [70] to extract the frequent patterns. The FP-growth algorithm utilizes a divide-and-conquer method based on partitions to mine the frequent patterns. The algorithm starts by examining the lowest nodes, i.e., the frequent patterns of length one. The algorithm proceeds to consider the conditional pattern base of these suffix nodes. Hence, it creates a conditional FP-tree consisting of the prefix paths that co-occur with the suffixes. The patterns that satisfy the threshold are generated from the conditional FP-tree. Thus, unlike the Apriori-based algorithms, FP-growth does not generate candidate sets, and with that, it can achieve better performance when considering the execution time. Furthermore, the algorithm scales better than the Apriori ones since it increases linearly with the number of items.

2.3 Stream Summarization

When confronted with massive amounts of data from a data stream, depending on the application, it may be beneficial to work only with a summary of the data. In our work on data stream summarization in restrictive environments, we will focus on an approach that forms item-based summaries based on a subset of items from the data stream. To determine the quality of a summary, we will assign a value based on a *utility function* to subsets of the data. Then, the values of different subsets will be compared to identify the best summary. A utility function can be any function that assigns a non-negative value to the subsets [172]. As the utility functions aim to find representative items, their function value should never decrease when new items are added to the summary (**monotone**), and adding items to a larger summary should have a smaller impact compared to a smaller summary (**submodular**). Most utility functions naturally fit into this class of being monotone submodular. In the following, we will introduce and analyze the different utility functions that can be used for finding core itemsets. In particular, we will consider two different *monotone submodular utility functions* which are based on active sets and exemplar-based clustering, respectively.

2.3.1 Active Set Utility Function

Active sets arise from Gaussian Processes, commonly used in nonparametric regression [137]. A Gaussian Process is essentially a joint probability distribution over a set of random variables, allowing it to model complex, nonlinear relationships in data. However, computing the predictive distribution for a new data point is computationally expensive because it requires inverting a kernel matrix that involves all the data points. This process scales poorly with large datasets. To address this computational challenge, predictions in Gaussian processes can be approximated using a smaller subset of the data, known as the *active set*, instead of the full dataset. The key to this approach is selecting an active set that still captures the essential structure of the data so that the results from the reduced subset closely approximate those obtained from the entire dataset. One effective method for selecting this active set is the Informative Vector Machine (IVM) [94], which identifies the most informative data points that contribute the most to the predictive distribution. By doing so, the IVM ensures that the performance remains accurate while significantly reducing the computational burden. Using IVM, the value of a subset S is formalized as

$$f(S) = \frac{1}{2} \log \det(I + \alpha K_{S,S}) \tag{2.1}$$

with a scaling parameter $\alpha \in \mathbb{R}_+$ and I being the identity matrix of size $|S|$. For submodular function optimization, the crucial part is that IVM uses a kernel matrix K for a subset S of size n with

$$K_{S,S} = \begin{pmatrix} k_{e_1,e_1} & \dots & k_{e_1,e_n} \\ \dots & & \dots \\ k_{e_n,e_1} & \dots & k_{e_n,e_n} \end{pmatrix}$$

where the kernel function k measures the similarity between all pairs of points in S. A popular choice for k is the RBF-kernel

$$k_{e_i,e_j} = \exp(-\frac{1}{2 \cdot l^2} \cdot ||e_i - e_j||_2^2)$$

with a normalization for the dimensionality d of $l = \frac{1}{2\sqrt{d}}$. The RBF is a positive definite kernel with normalized entries [144]. Thus, it can utilize the optimizations of Buschjäger et al. [24] for approximating the optimal value.

Maximizing the IVM then means to maximize the determinant, i.e., to have many values near 0 on the off-diagonal in matrix $K_{S,S}$ since those values represent the similarity between distinct elements. Since k_{e_i,e_j} measure the similarity of entries e_i and e_j, IVM provides a way to measure the dissimilarity between items of the subset. Consequently, this is exactly what is required from a utility function, to create a summary with maximized diversity.

The IVM was shown to be monotonic and submodular by Seeger [145]. An advantage of using IVM is that the value of the utility function only depends on the selected active set and not on the rest of the data. For this reason, the IVM is a popular choice for submodular function optimization in a streaming setting [12, 23, 24, 84].

2.3.2 K-medoids Utility Function

The k-medoids problem aims to build clusters around exemplars from the set of data points [83]. To find exemplars, the method works by minimizing the sum of pairwise dissimilarities. To compute distances, k-medoids requires a non-negative distance function $d : S * S \to \mathbb{R}$ between the data points. The cost function is then expressed as the sum of the distances to the closest exemplar considering all data points. Formally, the cost function is expressed using the following equation.

$$L(S) = \frac{1}{|V|} \sum_{v \in V} \min_{e \in S} d(e, v) \tag{2.2}$$

The cost function $L(S)$ depicted in Equation 2.2 can only decrease when new items are added. Thus, the function presented in this form is not yet a monotone submodular function. To make it submodular, as detailed in related work [63], an auxiliary element e_0 is introduced. This element is selected considering all points as the biggest distance to any given data point. Considering this addition, a submodular function can be created by decreasing the loss function L when adding the exemplars to a set that consists of only e_0. Formally, the function is expressed as follows.

$$F(S) = L(\{e_0\}) - L(S \cup \{e_0\}) \tag{2.3}$$

The function $F(S)$ depicted in Equation 2.3 can be maximized only when $L(S \cup \{e_0\})$ is minimized. This satisfies our requirement since this is exactly what minimizing the loss function $L(S)$ shown in Equation 2.2 does. Hence, $F(S)$ becomes a monotone function since the minimal distance to an exemplar can only decrease when adding new exemplars. For submodularity, let us consider adding an exemplar e to the sets A and B with $A \subseteq B \subseteq V$, where e is the nearest exemplar for $V_e \subseteq B$. We need to show $F(A \cup \{e\}) - F(A) \geq F(B \cup \{e\}) - F(B)$.

$$
\begin{aligned}
F(A \cup \{e\}) - F(A) = L(\{e_0\}) - \\
L(A \cup \{e\} \cup \{e_0\}) - (L(\{e_0\}) - L(A \cup \{e_0\}) \quad &(2.4a)\\
= L(A \cup \{e_0\}) - L(A \cup \{e\} \cup \{e_0\}) \quad &(2.4b)\\
= \frac{1}{|V|} \sum_{v \in V} \min_{e \in A \cup \{e_0\}} d(e, v) - \frac{1}{|V|} \sum_{v \in V} \min_{e \in A \cup \{e\} \cup \{e_0\}} d(e, v) \quad &(2.4c)\\
= \frac{1}{|V|} \sum_{v \in V_e} \min_{e \in A \cup \{e_0\}} d(e, v) - \frac{1}{|V|} \sum_{v \in V_e} d(e, v) \quad &(2.4d)\\
\geq \frac{1}{|V|} \sum_{v \in V_e} \min_{e \in B \cup \{e_0\}} d(e, v) - \frac{1}{|V|} \sum_{v \in V_e} d(e, v) \quad &(2.4e)\\
= F(B \cup \{e\}) - F(B) \quad &(2.4f)
\end{aligned}
$$

The submodular utility function F again formalizes an intuitive idea of utility. A set of exemplars with low loss represent the data well, as they minimize the distances to the data points. As demonstrated in the above equation, this can be precisely achieved by maximizing F. However, in practice, when working with data streams, using the utility function F is not directly possible. As visible in Equation 2.2, the distance to all data points is needed for computing the loss. When working with data streams, accessing all items or even storing them for later computation is not a viable option since data streams are potentially unbound. Consequently, the loss needs to be computed by an appropriate approximation without access to all the data. However, since the function is *additively decomposable* [116], it can be applied in a streaming setting. A monotone submodular function f is additively decomposable if f can be decomposed into sums of another monotone submodular function such that $f(S) = \frac{1}{|V|} \sum_{e \in V} f_e(S)$. The function f is then the average over all $f_e(S)$ and the average can be approximated with random samples. Thus, in our scenario, $L(S) = \frac{1}{|V|} \sum_{v \in V} \min_{e \in S} d(e, v)$ can be approximated by

$$
L_W(S) = \frac{1}{|W|} \sum_{v \in W} \min_{e \in S} d(e, v)
$$

where W is a set of uniform samples from V. In a streaming setting, the samples can be gathered continuously by using sampling techniques such as *reservoir sampling* [157] and used for computing F.

2.4 Learned Models

Learned models are used in various applications including but not limited to computer vision, text processing and generation, health data analysis, and also in the field of information systems. In our work, we make use of learned models for the tasks of cardinality estimation and approximate query processing. Machine learning models can be categorized into supervised and unsupervised models depending on the available training data.

Supervised models are trained on input data that includes so-called labels, i.e., the correct output [64]. During training, the supervised models learn how to predict the target data associated with the values of the input data. Once trained, the model can be used to predict the output for new input data. *Unsupervised models* are trained by using training data that does not have any information about the correct output, i.e., the training data is unlabeled [64]. During the training phase, the model tries to learn the underlying distribution of the input by identifying commonalities and patterns in the data. In our work, for the tasks of cardinality estimation and approximate query processing, we apply unsupervised models that learn directly over the data. In particular, we make use of Bayesian networks and deep autoregressive models.

2.4.1 Bayesian Networks

Bayesian networks are probabilistic graphical models that can approximate the joint probability distribution of a dataset [58, 118, 129]. A Bayesian network is a directed acyclic graph where the nodes of the network represent random variables, i.e., the attributes of a relation. The edges between the attributes denote a parent child relationship such that an attribute A_i is directly dependent on its parent attributes. Bayesian networks are able to efficiently represent conditional independence between groups of attributes such that every node in the network is conditionally independent of the nodes that are not descending from it, given the parents of the node [118]. As an example, if two nodes A_1 and A_2 are connected through an edge directed from A_1 to A_2, then this represents the conditional distribution $P(A_2|A_1)$. A Bayesian network is then a product of all the represented conditional dependencies. Formally, that can be represented by the following equation.

$$P(A_1, \ldots, A_n) = \prod_{A_i \in A} P(A_i|Parents(A_i))$$

`Parents`(A_i) represents the parent nodes of node A_i. The conditional probability distribution $P(A_i|Parents(A_i))$ represents the distribution of the values of the attribute A_i given any assignment of the values to the parents of A_i. This way of representing the joint distribution in a directed acyclic graph requires drastically less parameters than an unstructured joint distribution [118].

An example of a Bayesian network informing whether a fruit is tasty based on the size is depicted in Figure 2.4. The conditional probability distribution for every node, depending on the parents, is depicted in the tables accompanying the nodes. Since `fruit` and `size` do not have parent nodes, the probabilities for their values are shown.

In real-world scenarios, the structure of a Bayesian network will not be known but instead would have to be identified from the data. Examining all the possible architectures for a dataset has been shown to be an expensive process [34]. However, there exist approximate structure learning methods that have efficiently been applied in related work [33, 68, 69]. The Chow-Liu [33] structure learning algorithm works by only considering dependencies between two attributes. Consequently, the learned structure is a tree where every node will have one parent.

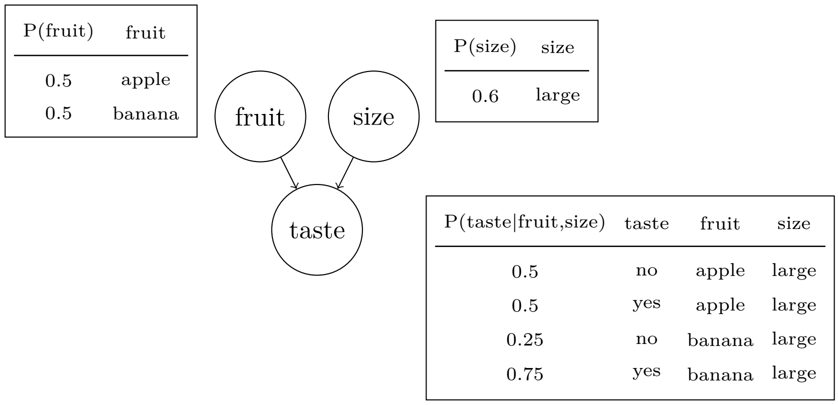


Figure 2.4: Example Bayesian network.

In the context of cardinality estimation and approximate query processing, to build the Bayesian network, every column of a table will represent a node in the network. To build the conditional probability distributions, we first require the unique values per column and, second, the individual tuples to compute the value dependencies. The values of attributes that consist of continuous values will be represented using binning. Thus, we split the value range into bins, and the conditional probabilities will be computed with respect to the individual bins. Naturally, this will affect the accuracy of the estimation.

2.4.2 Deep Autoregressive Models

Autoregressive models [118] are unsupervised models that can estimate a joint distribution $P(x)$ from a set of samples. They represent feed-forward models that can predict values based on past values. More specifically, for a predefined attribute ordering, the autoregressive property is that the output of the model contains the density for each attribute conditioned on the values of all preceding attributes. Given as input $x = [x_1, x_2, \dots x_d]$, the autoregressive model produces d conditional probabilities. The product of the conditionals results in the point density $P(x)$.

Formally, for $x = [x_1, x_2, \dots, x_d]$, the probability is calculated as:

$$P(x) = P(x_1)P(x_2|x_1)\dots P(x_d|x_1,\dots,x_{d-1}) = \prod_{d=1}^{D} P(x_d|x_{<d}) \tag{2.5}$$

The deep autoregressive model uses point density estimates $P(x)$ and conditional densities $P(x_i|x_{<i})$ obtained through various network architectures, including masked multi-layer perceptrons [43, 57] and masked self-attention networks [156].

In the context of relational databases, Naru [167] considers that each input x corresponds to one tuple of a table. Thus, the deep autoregressive model as input has all the tuples of the considered relation and, as output, produces the estimated probability distribution. During training, the model learns the

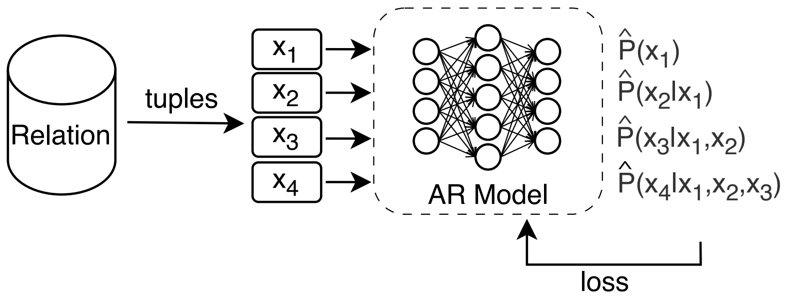


Figure 2.5: Overview of the framework Naru [167] for a table consisting of four attributes.

parameters by minimizing the cross-entropy loss [114,166,167] between the data distribution and the estimated distribution. Formally, the loss is represented by the following equation.

$$L(P, \hat{P}) = -\frac{1}{|R|} \sum_{x \in R} P(x) log \hat{P}(x) \tag{2.6}$$

To optimize the model parameters, the loss is minimized through backpropagation using stochastic gradient descent [64, 114].

Encoding and Decoding

The autoregressive models capture complex dependencies in relational data by utilizing information masking during the encoding process. By doing so, the proposed architecture Naru, creates a model where every attribute of the relation depends on all the values of the preceding attributes, respectively capturing all the values of an attribute. Encoding and decoding strategies are tailored to the size of the domain, with *one-hot encoding* for small and *embeddings* for larger domains. During encoding [114, 166, 167], for each attribute of the relation $X_i \in R$, the domain values are mapped to integers in the range $[0, |X_i|)$. For example, if the domain of column X_i is $\{Aachen, Mainz, Berlin\}$ then the values will be mapped as $Aachen \rightarrow 0$, $Mainz \rightarrow 2$, and $Berlin \rightarrow 1$, following the lexicographical ordering of the values. For every attribute X_i, the output layer will then have an $|X_i|$ dimensional vector representing the attribute distribution $P(x_i, x_{<i})$. One such distribution for our example can be $\{0.2, 0.3, 0.5\}$. However, for highly heterogeneous datasets, since the size of the deep autoregressive model is directly impacted by the input and output dimensionality, the size of the model will grow linearly with the number of unique values in the attributes. Hence, to overcome this problem, relevant related work [39,166] employs per-element compression with embedding for further reducing the space. Such compressions split the highly heterogeneous attributes into several subattributes so that the total dimensionality of the encoding is drastically reduced. The outlook of a deep autoregressive model for a relation consisting of four attributes is depicted in Figure 2.5.

Querying

Once the deep autoregressive model has been trained, in our work, we employ it for cardinality estimation and approximate query processing. As the model can, by construction, return point densities for a tuple, it is directly applicable for queries involving equality predicates. However, for range predicates, enumerating all ranges of the query for the table attributes is impractical. Thus, the previous work Naru [167] facilitated querying of ranges by the introduction of *Progressive Sampling*. Progressive sampling is an unbiased estimator for range predicates that focuses on high-density regions by employing an iterative algorithm. Through this method, a sample s from the relation R is created by sampling in iterations. More precisely, for every attribute, by following the fixed ordering, a sample is created by considering the distribution of the preceding attribute. Thus, the method consults the autoregressive model to create samples from more meaningful regions based on the query ranges. Additionally, an optimization known as wildcard-skipping allows for fast execution of queries over a subset of the columns.

Performing the progressive sampling, although required for answering range queries, imposes additional overhead on the estimation time. In our work on cardinality estimation, we circumvent the need for progressive sampling by combining the autoregressive model with a grid index. Hence, the model performs point density estimations and can provide answers in drastically less time.

Chapter 3

Related Work

3.1 Local Join Processing

When considering traditional relational database management systems, there has been an abundance of work on relational join processing, like hash joins, variants of (index) nested loop joins, and sort-merge joins [14, 41, 89]. Many of these approaches can be considered textbook knowledge nowadays. More recent work exploits modern hardware for making the traditional approaches more efficient, also taking into account parallel executions of the considered algorithms [9,13,79,87,93,110]. However, when applied to schema-free data where the attributes on which the join will be performed are unknown, approaches designed for relational data have a significant drawback. When not knowing the join attribute, the hash join and the index-based nested loop join do not benefit from indexing structures and a meaningful order is not derivable to support, for instance, the sort merge join.

With the widespread use of JSON data, the past years have witnessed various approaches that try to incorporate JSON support into existing relational query engines [44, 151]. More specifically, in their very recent work, Durner et al. [44] propose to group JSON documents by their available attributes in so-called tiles and describe how the query processing is conducted over tiles in a RDBMS. The way tiles are formed does not seem compatible with the notion of natural joins. Bourhis et al. [18] focus on establishing a theoretical framework for JSON and propose a lightweight query language for traversing JSON documents. JSONiq [51] and SQL++ [125] are additional query languages for processing JSON documents. Although they support a variety of functions on top of JSON documents, they do not investigate nor propose efficient algorithms for joining JSON documents. In contrast to such efforts, NoSQL systems like MongoDB and CouchDB were specifically tailored to the requirements imposed by JSON and large datasets, but are limited in the query language and, in particular, joins. Joins are supported by Apache Spark SQL and can also be realized in MapReduce frameworks. However, the challenge of finding suitable keys to partition tuples in the case of schema-free natural joins remains untouched. Another branch of research considers processing so-called raw data in form of CSV or JSON files [8, 142]. Such approaches refrain from classical data indexing and can be considered tools to enable the first steps in data exploration, cleaning, and transformation, but with no or limited support of joins, in particular, natural joins over schema-free data.

3.2 Join Processing in Streaming Environments

Joining data streams in a distributed environment is a topic that has been widely researched and is most prominently visible in work that focuses on performing traditional SQL-like joins over tuples defined in a specific window. Kang et al. [81] analyze several stream join algorithms and inform the most suitable approaches for different scenarios. Recently, the idea of a join-matrix model [149] has been revisited and reused for distributed join processing for both streaming [46] and map-reduce applications [124]. This model represents a join operation as a matrix, where dimensions correspond to the relations that need joining and the matrix cells represent the join output result. This approach does not scale well and suffers from high memory consumption due to the constant data replication. BiStream [104], which outperforms the join-matrix model, is a scalable and memory-efficient distributed stream join system. It is based on a novel join model, referred to as join biclique, which organizes the processing units of clusters as a bipartite graph. Although it outperforms existing systems, the performance of BiStream degrades on skewed data and it is mainly developed for dealing with equi joins. Another distributed stateful stream joining system, called Photon [11], focuses on joining web search queries and clicks on advertisements by using a unique identifier present in both events while guaranteeing exactly-once semantics and allowing both unordered or delayed streams.

Unlike the MapReduce implementations of equi joins [5, 16], TimeStream [134] combines both MapReduce-style batch processing and streaming database systems. It exploits tuple dependencies to perform joins. The Pipelined State Partitioning (PSP) [159] approach focuses on distributed processing of generic multi-way joins with window constraints by partitioning the states into disjoint slices and distributing the states using the created slices. Both PSP and TimeStream suffer from large communication overhead. Several algorithms also exist for a multi-core and main-memory environment [56, 139, 153].

ApproxJoin, introduced by Quoc et al. [135], is a distributed stream joining approach that provides approximate results. The approach is realized by using Bloom filter sketching, which minimizes the network traffic, and stratified sampling through which the statistical properties of the join output are preserved. D-Stream, introduced by Zaharia et al. [168], splits the streaming computation into deterministic batch computations and, for each, performs a MapReduce job. However, D-Stream is not applicable in our streaming join processing scenario since by grouping the data into small batches, candidate tuple pairs for joining may miss each other. Hence, this approach can only provide approximate join results, like ApproxJoin. Even though many of the approaches are scalable, efficient, and perform well for large datasets, our work on join processing majorly differs in focus, which is performing stream joins when the join parameters are not given or known.

3.3 Partitioning in Streaming Environments

Another thoroughly analyzed research field when considering stream processing and a subproblem of distributed stream joins is data partitioning. A common observation is that approaches based on hash and range partitioning [26, 82, 105, 174] suffer from poor load balancing in the presence of skewed distribution or in cases where one key appears more than the others. Additionally, hash partitioning on several keys may result in scenarios where the joinable documents are not collocated on the same machine. The shuffle partitioning [82] blindly assigns tuples to machines. Thus, it is inadequate for our considered stream join scenario since it will not place the same keys on the same machines. Since a document can be represented as a graph, graph partitioning methods are also applicable. The work presented by Alvanaki and Michel [10] continuously creates and maintains partitions of Twitter hashtags to compute hashtag co-occurrences for trend analysis. We thoroughly analyze and compare with the proposed partitioning algorithms. The Kernighan-Lin algorithm [86] minimizes the edge cut by moving vertices between partitions. The work presented by the authors of [27, 86, 141] is based on the Multilevel Graph Partitioning [75] where they improve the partitioning by combining it with different heuristics. In a dynamic environment, these approaches are computationally expensive due to the data changes as the stream flows, resulting in a partition that is valid only for a short time.

3.4 Summarization of Data Streams

Although different notions of *core sets* exist in related work on data streams [77, 115], the terminology in this work is based on the *core item* and corresponding *core item sets* that are used by Zhao et al. [172]. The core items are representative items chosen from a data stream. The problem of *core items tracking* is to continuously maintain core items in a streaming setting. Their approach is based on probabilistic decay. Since this work deals with a setting where the more recent data is not necessarily more important, but instead, all items within a window are equally important, these approaches are not considered in the comparisons.

Sieve-Streaming [12] is a popular algorithm for submodular function maximization of data streams and the basis for most of the related research as well as the approach introduced in this thesis. Similar to Zhao et al. [172], the authors formalize the problem of summarizing a data stream as the maximization of a submodular set function subject to a cardinality constraint. They reduce the more general utility function to a submodular function, adding the property of diminishing returns, and argue that this is fitting for most choices of utility functions. The algorithm, unlike some previous approaches [97, 122], uses a single pass over the data, which is essential for streaming applications. This approach achieves the best possible approximation for this setting [49]. Sieve-Streaming++ [84] is motivated by the observation that Sieve-Streaming unnecessarily maintains sieves that cannot achieve the best utility value over all sieves anymore. These sieves can be safely discarded and replaced by sieves based on the new lower bound. Dynamic Sieve Streaming [24], unlike the previous approaches, is explicitly aimed at a distributed IoT setting. It proves improved upper and lower bounds to be used for the active set utility function, which is a popular choice for a submodular utility function [12, 23, 84] and one of the utility functions that we use in our experiments. In the approach, sieves that are known to be suboptimal are discarded and replaced by sieves with new thresholds. However, the experiments were not done in an IoT-representative setting but on a machine with higher processing power and memory size. Moreover, the improvements are specific to only the active set utility function, and we aim to compare approaches on multiple utility functions. Thus, this approach will not be included in our experiments. The previous algorithms do not weaken the theoretical guarantees of Sieve-Streaming. In contrast, ThreeSieves [23] ignores the theoretical worst case and aims for better practical performance. ThreeSieves maintains only one threshold, drastically reducing memory cost while, also, enabling the possibility dynamically changing that threshold over time.

The related research on approaches that focus on submodular function optimization is vast [37, 102, 103, 122, 172]. However, these approaches will not be explored in more detail, as they either do not offer significant improvement over the mentioned approaches or do not apply to our edge streaming setting.

3.5 Cardinality Estimation and AQP

3.5.1 Non-learned Cardinality Estimation

The recent advancements in learning–based cardinality estimation allude to a clear separation in the field between non-learned and learned estimation approaches. The *non-learned approaches* can be further categorized into traditional approaches which are based on statistics gathered typically through histograms [21, 35, 40, 78, 92, 117, 132] and sampling–based approaches [32, 47, 92, 96, 99, 160, 173] that, naturally, rely on samples gathered from the relations to estimate the query cardinality. The sampling–based approaches struggle with efficiency since they are often slower than non-sampling approaches [71]. Typically, they are developed for estimating equality joins [92]. The approaches that rely on histograms are widely adopted and are still the standard for estimation in modern database management systems such as PostgreSQL [133], SQL Server [161], and MySQL [119]. Although they produce estimates fast, they fail to capture the correlation between the attributes and they assume a uniform key distribution. Thus, when working with complex queries or skewed data distributions, these approaches seriously misestimate the result.

3.5.2 Learned Cardinality Estimation

Approaches based on machine learning models try to improve the cardinality estimates in different ways, e.g., by relying on prior query monitoring [150], multi-set convolutional networks [88], simple deep learning models [127], pure data-driven models [76], local-oriented approaches [162], containment rates [74] and other neural network architectures for relational operators [106] or multi-dimensional range predicates [45]. Others [90, 111–113, 126] use reinforcement learning for optimal plan generation. Naru [167] achieves accurate cardinality estimates with autoregressive models and a Monte Carlo integration technique. NeuroCard [166] builds upon Naru by introducing per-column compression to reduce the memory and learns over samples from the join to enable efficient join cardinality estimation. Hasan et al. [73] suggest both autoregressive and supervised models for accurate estimates. LMKG [39] uses autoregressive models with per-column compression and supervised models to estimate cardinalities over knowledge graphs. For the comparison, we utilize their compression for our Grid-AR estimator and the autoregressive model that we use for approximate query processing. Flat [175] uses factorized sum split product networks for accurate and fast cardinality estimation. Their results further underpin the accuracy of deep autoregressive models and the need for faster estimation. BayesCard [71,165] creates an ensemble of Bayesian networks to model the table's distribution and employs progressive sampling and variable elimination inference algorithms for multi-table join queries. Differently, Flow-Loss [121] is a novel loss function for training cardinality models that focuses on optimizing estimates critical for query optimization, resulting in improved runtimes despite potentially worse estimation accuracy. Meng et al. [114] combine Gaussian mixture models and autoregressive models to improve estimation for continuous attributes. Their model, though similar to our proposed Grid-AR model, demands considerably more training time and complexity. FactorJoin [164] formulates the estimation of join cardinality as an inference involving only learned distributions over single tables.

3.5.3 Approximate Query Processing (AQP)

Sampling-based approaches for AQP have been widely researched, dating back several decades [30, 35]. These approaches come with limitations that are well known. Many sampling-based approaches use information about query workloads to produce better results [6, 29, 42]. There also exist approaches that create samples online [80, 99, 163]. Wander join [99] creates estimates by modeling the tables as a join graph and performs random walks through the graph. VerdictDB [128] is an AQP system that supports approximate query processing of general ad-hoc queries. It works over so-called scrambles, which represent samples from the original data.

Various approaches use precomputed aggregates or a combination of precomputed aggregates and sampling [54, 65, 101, 130]. Liang et al. [101] combine precomputed aggregates with stratified sampling for producing query estimates. The proposed indexing structure PASS represents a hierarchical partitioning of the dataset such that for every partition in the tree, it stores the `MIN`, `MAX`, `SUM`, and `COUNT`. The leaf nodes store a uniform sample of the data from the respective partition. The approach is limited to only answering aggregation queries without joins. Similarly, AQP++ [130] precomputes the results to numerous aggregation queries. It determines a query subsumption relationship to match a new query to an existing one and then uses uniform samples to approximate the gap. AQP++ can also only answer queries that do not have joins.

There also exist approaches that apply *machine learning for approximate query processing* [39, 76, 91, 109, 154, 158, 166, 167, 170]. DeepDB [76] learns directly over the data by using relational sum product networks to efficiently capture the database characteristics. As explained, Naru [167] and related works [39,166] uses deep autoregressive models and train directly over the data for selectivity estimation. Although for our work on approximate query processing (Chapter 6) both underlying models can be considered as bubble summaries, we make use of the deep autoregressive models.

It is essential to point out that although our work on approximate query processing aims at estimating results for aggregation queries, we want to accomplish this by moving the computation from tuples to groups of them, the so-called tuple bubbles. In the future, many approaches that support joins can be employed to store per-tuple-bubbles models.

Chapter 4

Natural Joins in a Streaming Environment

4.1 Introduction

In this chapter, we consider computing natural joins over schema-free JSON [3] documents that arrive in a data stream. We will do so for a scale-out architecture, where the load of join computation is spread across multiple nodes that perform the actual join evaluation while being resource efficient and guaranteeing the exact join result. As data, we consider application-tailored data formats without fixed schema due to the ever-increasing number of newly created information that is typically generated from various sources. The content of this chapter are based on our publications at ICDE 2020 [60] and the SCDM workshop co-located with IEEE Big Data 2021 [61]. We will consider JSON documents because JSON is, arguably, one of the most visible data formats, which eliminates the need to force data into relations, is designed to be human-readable, and has been accepted across various platforms as the choice for data exchange. For instance, Twitter provides access to its public tweets via interfaces that deliver JSON data and platforms like geojson.io, likewise, enable uploading JSON data containing geo-coordinates for automatically placing them on a map. With the wide acceptance of JSON, database systems tailored to handling JSON content are flourishing as part of the still ongoing NoSQL movement—while traditional vendors and communities around traditional RDBMS are extending their relational storage engines and query syntax to deal with JSON content, too. The flexibility JSON allows also comes with downsides when it comes to applying traditional query processing techniques to arbitrarily shaped data. The ubiquitously used relational join vastly benefits from fixed-schema relational data, to create appropriate hash tables or sorting orders, or using index structures across primary-foreign keys, for improved performances.

Although performing joins in a distributed environment is a deeply investigated topic, to the best of our knowledge, the main focus is on performing traditional SQL-like equi or theta joins where the join parameters are either known in advance or provided at query time. We address the problem of computing *natural joins* in a distributed environment on top of schema-free JSON documents. **If two documents share at least one attribute-value pair while at the same time they do not have any conflicting values for the same attributes, they will be provided as resulting joinable documents.** For producing the correct join results, we rely on a partitioning strategy that ensures that potentially joinable documents will end up at the same machine, where they are subsequently joined. The final join result then enables the analysis of documents that contain complementary information, without knowing the join predicate in advance.

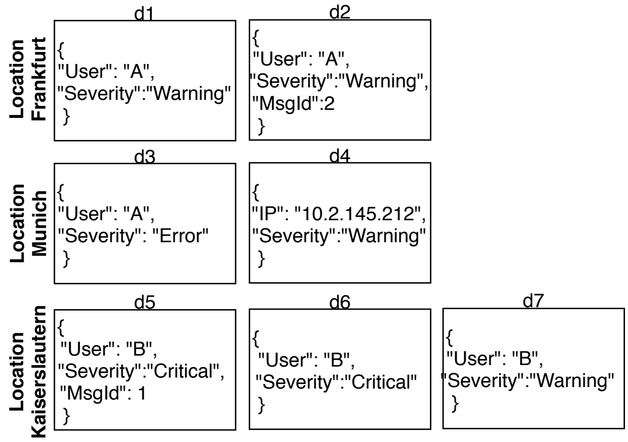


Figure 4.1: Example for joinable documents. The documents are grouped based on the location where they are created.

Consider the sample JSON documents displayed in Figure 4.1, showing an excerpt of a company's server access log, with information about user logins and file accesses. Analyzing the stream of documents can provide beneficial insights for the company, such as identifying an attack on the servers, for instance, by detecting a constant failed login attempt on one location or identifying a user with constant failed access to files which may be a result of a virus-infected work station. The importance of the partitioning strategy is emphasized by the documents in Figure 4.1. If the documents are partitioned based on their entire set of attribute-value pairs, the correct join result will not be produced since joinable documents will end up in different partitions. Additionally, if they are partitioned based on an arbitrary attribute, the unequal occurrences of the attribute's values in the documents can contribute to creating partitions that differ greatly in the number of documents that they need to process. To underpin the problems traditional join processing techniques would face, consider bringing these seven documents into the relational model. The corresponding database table would require five columns, and even for this small example, many data entries would be NULL. With the simplifying assumption that there is a common attribute in all the documents, we could, in fact, employ a simple hash index, but this would shift the problem of efficient join result identification to the groups of documents that share the same attribute value. In contrast, we propose a native approach to handle natural joins over schema-free data.

4.2 Contributions and Outline

- We propose a partitioning algorithm based on the concepts of association analysis. The created partitions lead to superior performance and scalability compared to existing approaches.
- We discuss scalability limitations of partitioning approaches due to attributes with low value variety and how to circumvent this by deriving synthetic attribute-value pairs based on original ones.

- We introduce two novel algorithms for computing natural joins over schema-free documents, based on frequent pattern (FP) trees.
- We detail the possible indexing strategies when considering natural joins on multiple local datasets.
- We discuss a practical realization of our solutions in Apache Storm.
- We report on an elaborate experimental study using real-world as well as synthetic datasets and compare our approach to competitors.

The remainder of this chapter is organized as follows. Section 4.3 sketches the high level scale-out architecture and gives an overview of the important aspects concerning data partitioning. The two main tasks, data partitioning and local join computation are presented in Section 4.4 and Section 4.5, respectively. While in Section 4.5.3 we address the handling of joins beyond self-joins at the local machines. Section 4.6 discusses how the system reacts to documents that do not match any partition, how and when partitions are updated, and how to avoid limited scalability in presence of Boolean and other attributes of small value domain. Section 4.7 reports on the results of the experimental evaluation, while, eventually, Section 4.8 summarizes the chapter.

4.3 Preliminaries and Proposed Architecture

Our goal is to assign partitions composed of attribute-value pairs to m machines. The incoming JSON documents are forwarded to the machines based on these partitions. As an example, consider the documents from Figure 4.1 and assume that there are two machines. One possible partitioning could be:

- $pr_1 = \{\text{"}User\text{"} : \text{"}A\text{"}, \text{"}Severity\text{"} : \text{"}Warn.\text{"}, \text{"}Severity\text{"} : \text{"}Err.\text{"}, \text{"}MsgId\text{"} : 2, \text{"}IP\text{"} : \text{"}10.2.145.212\text{"}\}$
- $pr_2 = \{\text{"}User\text{"} : \text{"}B\text{"}, \text{"}Severity\text{"} : \text{"}Crit.\text{"}, \text{"}MsgId\text{"} : 1\}$

A document matches a partition if both share at least one attribute-value pair. This is important, as it reflects the requirement that two documents that even share just one attribute-value pair can potentially be joined. Based on the sample partitions, the documents d_1, d_2, d_3, d_4, d_7 will be assigned to the first machine and the documents d_5, d_6, d_7 will be assigned to the second machine.

The number of documents assigned to any of the machines represents the **load**. If we assume that the documents do not change over time, the machines responsible for pr_1 and pr_2 will have a load of 62.5% and 37.5% respectively. Ideally, all the machines should operate on an equal load which will mean that they will require the same amount of resources and processing power to perform their computations. One can observe that the sum of the documents assigned to partitions pr_1 and pr_2 is larger than the actual number of documents. This is a result of replicating document d_7. This **replication** cannot be avoided since the document is joinable with both documents from pr_1 and pr_2. The number of replicated documents will represent the communication overhead because it represents the number of messages that need to be sent to every machine. Mutually exclusive partitions that have approximately equal load cannot always be achieved due to data characteristics. For that reason, the partitioning algorithm

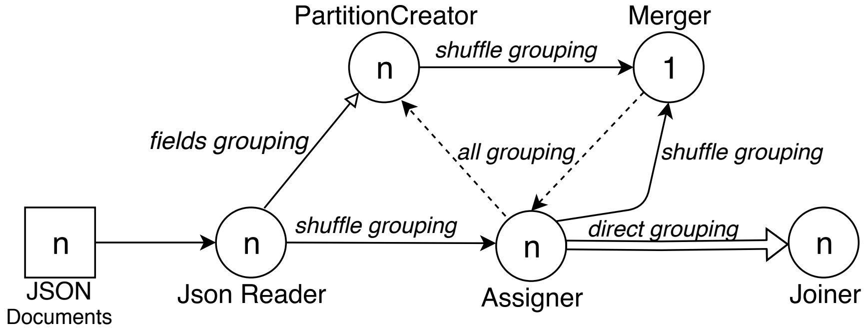


Figure 4.2: Stream join Apache Storm topology.

should aim to minimize the replication of documents for lower communication overhead and create partitions that distribute the load more equally among the machines. We will detail the proposed partitioning algorithm in Section 4.4.

Documents that are forwarded according to the partitioning are subsequently matched against previously arrived documents and stored for documents in the future—until the window dictates the eviction. To ensure efficient matchmaking, Section 4.5 introduces two join algorithms tailored to the peculiarities of schema-free data, using FP-trees.

4.3.1 Architecture

The architecture we realized is a rather general topology comprising multiple nodes within a compute cluster. We decided to use Apache Storm [2], a mature and widely applied platform for streaming applications. Of central importance are those nodes that perform the actual join over the documents they get assigned. This assignment is done based on partitions, which are computed upfront and are continuously maintained. The task of computing partitions is parallelized, by creating a partitioning from disjoint samples of the data stream and, then, combining the partitioning into a final set of partitions. This resembles the generic topology presented by Alvanaki and Michel [10] for the task of counting hashtag co-occurrences. Figure 4.2 depicts the topology. For every component, there can be either one (1) or multiple (n) instances or tasks and the lines depict the different subscription methods used for distributing the data among the instances of the components. We can observe the following components:

1. **Partition Creator**: creates partitions based on the attribute-value pairs of the currently seen documents. There can be multiple Partition Creators for sharing message load and computation.

2. **Merger**: receives partitions from the Partition Creators and merges them into a global set of partitions. It also updates the partitions. To satisfy the requirement of having a consistent set of partitions, only one instance of the Merger can be created.

3. **Assigner**: simple dispatcher that forwards documents to the individual Joiner instances according to partitions. Informs the Merger once a

document with unseen attribute-value pairs appears and initiates the re-computation of the partitions once the quality of the partitions is below a pre-defined threshold.

4. **Joiner**: responsible for computing the actual join between documents it got assigned.

4.4 Partitioning Based on Association Groups

Distributing the documents evenly among the machines with acceptable replication factor and load balance, urged us to develop a scalable and efficient partitioning approach. The proposed partitioning approach uses the attribute-value pairs of the documents to create the required partitions. By analyzing the patterns of occurrences of the attribute-value pairs, the documents are partitioned based on the meaning rather than their set of attribute-value pairs. In the following section, we present a partitioning algorithm based on *association rules*.

The algorithm is based on the observation that the occurrences of the elements within the documents are not arbitrary. The elements can depend on one another and regularly follow the same pattern of occurrence. Some elements constantly appear together, in groups, which we call *equivalence groups*. Additionally, some attribute-value pairs appear only when a specific attribute-value pair appears and not alone. We call the latter *association groups*. A clear connection to association analysis can be observed, but we do not adhere to the same principles for finding the association groups. Unlike the traditional association rule mining [152], we do not use support and confidence to eliminate groups that occur by chance. If the elements form an association in at least one document, they will be considered as a valid association. The reasoning behind this choice is that we want our approach to provide results in real-time without having high computational complexity. For that purpose, we make a trade-off between the actual importance of an association group and the execution time of the algorithm. Detecting frequent itemsets is not only computationally expensive but will also lead to incorrect results. Considering only groups of elements with support and confidence above some predefined threshold will lead to having many documents that will not be assigned to any partition. As a result, they will not be emitted to any node and our statement of providing all documents that can be joined with one another will not be met.

The equivalence groups are identified by determining which attribute-value pairs satisfy the *equivalence relationship*. The association groups are calculated from the computed equivalence groups by establishing the equivalence groups that satisfy the *implies relationship*. We define the equivalence and implies relations as follows:

Definition 1 ***Equivalence relationship*** *Let* $S = \{d_1, ..., d_n\}$ *be the set of all documents and* $av_i = \{a_1 : v_1, ..., a_m : v_m\}$ *be the set of attribute-value pairs for the document* d_i*. The attribute-value pairs* $(a_i : v_i, a_j : v_j)$ *form an equivalence group eg if whenever* $a_i : v_i$ *appears in document* d_i *then also* $a_j : v_j$ *appears and vice versa.*

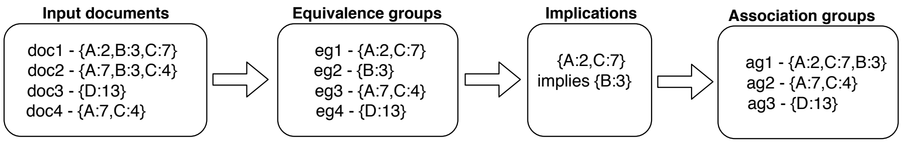


Figure 4.3: Finding equivalence and association groups.

Definition 2 ***Implies relationship*** *Let eg_i and eg_j be two computed equivalence groups. We say that eg_i implies eg_j if whenever the attribute-value pairs from eg_i appear in a document then also the attribute-value pairs from eg_j appear, but the attribute-value pairs of eg_j can appear individually without the attribute-value pairs of eg_i.*

Figure 4.3 represents an example of finding association groups. Based on the definition for the equivalence relationship, the algorithm determines that $eg_1 = \{A:2, C:7\}$ is an equivalence group since $A:2$ and $C:7$ always appear together in all the documents. Based on the equivalence groups it is determined that eg_1 implies eg_2 since in every document where $\{A:2, C:7\}$ appears also $B:3$ appears. By grouping together the equivalence groups that satisfy the *implies* relationship, the final association groups ag_1, ag_2, and ag_3 are produced.

4.4.1 Algorithm

The partitioning algorithm based on association groups is executed in two phases. In the first phase, the association groups are calculated from the attribute-value pairs of the documents and, in the second phase, the partitions are created based on the previously computed association groups.

Algorithm 1 Association Groups–Based Algorithm

Require: Set of documents D

$avInD = getDocsForAvPair(D)$ ▷ create equivalence groups
$EG = avInD.keys$
sort(EG) ▷ sort in ascending order based on #docs
$AG = \{\}$
for i **in** $EG.size$ **do** ▷ create association groups
  $ag_i = EG[i]$
  for $j = i + 1$ **in** $EG.size$ **do**
    if $EG[i]$ **implies** $EG[j]$ **then**
      $ag_i = ag_i \cup EG[j]$
      $EG = EG \setminus EG[j]$ ▷ remove implied equivalence group
    end if
  end for
  $l_i = |\cup_i d_i|,\ ag_i \in d_i$ ▷ compute load for association group
  $AG = AG \cup ag_i$
end for
return AG

The realization of the first phase is presented in Algorithm 1. The algorithm starts by creating a map referred to as $avInD$, where the key is a set of documents and the value is the attribute-value pairs that appear in that set of documents (Algorithm 1, Line 1). With the creation of the map, actually, the equivalence groups have been created since all the attribute-value pairs that appear in the same set of documents have been grouped together. The list EG is created by adding all the keys from the map $avInD$ (Algorithm 1, Line 2). The keys represent the sets of documents. The list of equivalence groups is sorted in ascending order based on the number of documents in every group (Algorithm 1, Line 3). The association groups are created by iterating over the list of equivalence groups EG and checking if the *implies relationship* is satisfied (Algorithm 1, Lines 5–15). If it is determined that equivalence group eg_i implies the equivalence group eg_j, then eg_j is removed from the list EG to avoid the creation of association groups with overlapping attribute-value pairs (Algorithm 1, Line 10). Additionally, for every association group ag_i, the load is computed by counting the number of documents in which the attribute-value pairs of ag_i appear (Algorithm 1, Line 13).

The required m partitions are formed by using the nonoverlapping association groups generated by Algorithm 1. The initially empty partitions are populated with the first m association groups that have the highest load. Once the initial m partitions have been created, in every new iteration the association group with the highest load is selected and it is assigned to the partition that has the least load. This procedure is repeated until all the association groups have been assigned to partitions. Through this technique of assigning the association groups, partitions with approximately equal load will be created. The idea for assigning sets to partitions was already introduced by Alvanaki and Michel [10]. We can apply the same idea in our partitioning approach since both the association groups and disjoint sets partitioning algorithms as output create sets with nonoverlapping elements.

Applying this partitioning algorithm in our distributed Apache Storm topology requires one more step, the creation of the consolidated association groups. If the complete partitioning algorithm is used in every PartitionCreator, incorrect partitions will be created. This is the case because PartitionCreators are oblivious of the whole set of documents on which the partitioning is performed. When they compute the association groups for their own set of documents, it does not mean that the same association groups will apply for the global set of documents. Thus, only the first phase of the partitioning algorithm (Algorithm 1) is executed by the PartitionCreators. The local association groups from every PartitionCreator will be emitted to the Merger that will have the responsibility to create the final, correct association groups and assign them to the m partitions.

To unify the association groups, the Merger first merges all the association groups that are a subset of another association group. For eliminating the duplicate attribute-value pairs, the Merger analyzes the association groups and if an attribute-value pair is part of two different association groups it is removed from the association group with more elements. Using the consolidated association groups the Merger populates the available m partitions following the approach explained above.

Table 4.1: Example set of documents and their reordered representation imposed by the global attribute order (right column).

ID	Attribute-Value Pairs	Ordered AVP
d_1	$\{a:3, b:7, c:1\}$	$\{b:7, a:3, c:1\}$
d_2	$\{a:3, b:8\}$	$\{b:8, a:3\}$
d_3	$\{a:3, b:7\}$	$\{b:7, a:3\}$
d_4	$\{b:8, c:1, d:1\}$	$\{b:8, c:1, d:1\}$
d_5	$\{b:9, c:2, d:1\}$	$\{b:9, c:2, d:1\}$

4.5 Local Join Computation

After the partitions are created and deployed in the system, potentially joinable documents are routed to the same compute node. The next step is to perform the actual join algorithm. For determining if two documents can be joined, we adhere to the join definition presented in Chapter 1.1: two documents are joinable if and only if they have identical values for the attributes they have in common—documents that do not share any attribute cannot qualify for the join result. In the following, we introduce two flavors of a novel join algorithm that uses the FP-tree [70] as a foundation for discovering all the joinable documents. As next, we first detail the construction of the FP-tree and then explain the two variants for performing the local join computation.

4.5.1 FP-tree Construction

The FP-tree, introduced by Han et al. [70], is an extended prefix-tree that stores data in a compact way. Initially, the FP-tree was introduced in the field of frequent pattern mining, but with small modifications, it can be also applied for storing attribute-value pairs of documents.

To build the FP-tree, we need to impose a strict ordering of the elements that will be stored in the tree. To accomplish this, we examine the set of documents and sort the *attributes* of the documents in descending order by counting the number of times that they appear in different JSON documents. When the number of documents in which two attributes appear is the same, the attribute with fewer unique values will have a higher priority in the ordering. As an example, consider the set of documents shown in Table 4.1. Based on the approach explained above, attribute b will have the highest priority in the ordering since it appears in the highest number of documents, i.e., five. Attributes a and c appear in the same number of documents so their position in the ordering is set according to their number of unique values. Because attributes a and c have one and two unique values, respectively, a will have a higher priority in the ordering. Ultimately, since attribute d appears in the least amount of documents the final strict ordering will be $b \to a \to c \to d$. The second column of Table 4.1 shows the reordered documents.

Once the ordering of the attribute-value pairs has been specified, we can proceed with the construction of the FP-tree. When the FP-tree is created, it can be directly utilized for identifying the set of joinable documents.

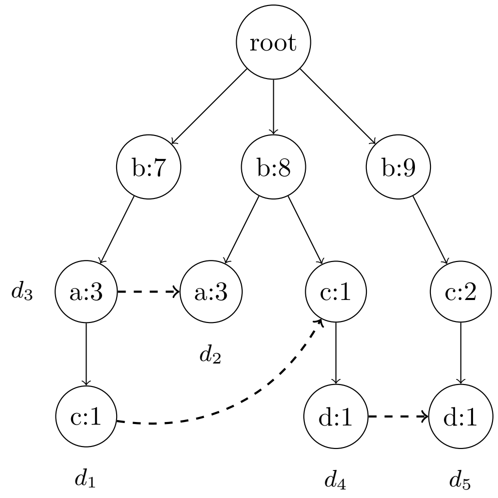


Figure 4.4: FP-tree for the documents in Table 4.1.

In the beginning, the FP-tree is empty and contains only the root node labeled with 'null'. The nodes in the FP-tree are identified by a label, which, in our scenario, is an attribute-value pair of a document. Additionally, the nodes in the FP-tree can have a pointer to the next node in the tree that has an equal label, i.e., represents the same attribute-value pair or, otherwise, null as a value of the pointer. Finally, to represent the considered JSON documents, we store a list of document identifiers (ids) that specify the documents to which the nodes of the branch belong. The list of document ids will be used for producing the documents that can be joined. To distinguish between different branches of the tree every branch carries a unique branch id. Together with the FP-tree, we create a header table which, as in the original paper [70], points to the first node labeled with a specific attribute-value pair.

The FP-tree for the example set of documents shown in Table 4.1 is depicted in Figure 4.4. In every iteration, the attribute-value pairs of the next considered document are added as branches in the tree. Following the imposed strict ordering, the attribute-value pairs of the document are selected one by one and added as direct children in the same branch. However, if the tree already has nodes labeled as the currently considered attribute-value pair, instead of creating a new node, the algorithm navigates down the branch until there is an attribute-value pair without any equally labeled node in the branch or until all the attribute-value pairs of the document have been covered. For document d_1, being the first, we create a new branch with its attribute-value pairs. For document d_3, the algorithm will detect that the root node has already a child labeled with $b{:}7$, and instead of creating a new branch, the algorithm navigates down the branch that starts with the node $b{:}7$. The next attribute-value pair of document d_3 is $a : 3$. The children of the node $b : 7$ are analyzed and it is determined that there is already a node labeled with $a{:}3$. Hence, the node $a{:}3$ represents the next starting point for the resumption of the algorithm. Because this is the last attribute-value pair of document d_3, the list of documents in the node $a{:}3$ is updated by adding the document id. The same procedure is applied to the remaining documents producing the FP-tree shown in Figure 4.4. The dashed arrows represent the links between the equally labeled nodes.

We eliminate the count stored in every node that was proposed by Han et al. [70] since we do not need to keep track of the number of times that the attribute-value pairs appeared within the documents. Instead, in selected nodes, we store a list of document ids that indicates the exact documents in which the nodes of the branch have occurred. With the creation of the FP-tree, the problem of iterating over too many documents is solved since the attribute-value pairs present in the same order in multiple documents will be stored only once. Additionally, with the helper links between the equally labeled nodes, a possibility for faster traversal of the tree is provided. The links between the equally labeled nodes and the header table will be created only for the first version of our join algorithm (Section 4.5.2) because the FP-tree will be traversed by navigating through the connected nodes.

To use FP-trees at the join processing nodes in our topology, we first have to compute an attribute ordering, which is done right after the partitions are created and the documents start getting assigned to the join processing nodes. The documents received at the *Joiners* are then tested against the stored documents and then added to the tree, to be matched with forthcoming documents. This means that the creation of the tree and the matchmaking is done in a batch fashion. In our current version, we investigate so called tumbling windows, which define nonoverlapping chunks of documents. This allows to evict the entire tree once the window tumbles. For sliding windows, tree updates or frequent tree evictions and rebuilds are required.

4.5.2 Join Algorithms

Having indexed the JSON documents with an FP-tree, we next delineate an approach that uses the constructed structure for identifying the joinable partners. We term our join algorithm *FPTreeJoin* and propose two different variants of the algorithm. The first one is more suitable when the documents consist of frequent attributes that have a large number of unique values, while the second version is appropriate when the most frequent attribute has a small number of distinct values. In the following, we will refer to the former as FPTreeJoin_v1 and the latter as FPTreeJoin_v2.

The need for different flavors of the join algorithm comes from the potential nature of the documents. For the FPTreeJoin_v1 algorithm, let us assume that same combinations of attribute-value pairs do not appear regularly in the documents and none of the attributes is present in every document. Consequently, we cannot draw any conclusions about the immediate children of the root node since they can all represent different attributes. Thus, to compute the join, we need to traverse the nodes of the FP-tree while efficiently pruning branches that do not hold joinable documents.

For the use of FPTreeJoin_v2, assume that one attribute with small number of unique values is present in all the documents. This can be a Boolean attribute that is part of every JSON document with either the value $True$ or $False$. Based on the strict order definition since this attribute appears in all documents and has the least distinct values, it will have the highest priority in the ordering.

Algorithm 2 FPTreeJoin_v1 - Part 1

Require: Set of attribute-value pairs *avP*, Node *node* from the FP-tree, List of branches to ignore *prunelist*

```
1:  function isCandidate
2:      if node.parent = 'root' then
3:          return true                                   ▷ branch with potential join partners
4:      end if
5:      pAvP = node.parent.avPair
6:      if node.parent.branchId in prunelist or
7:      (pAvP.attr ⊂ avP.attrs and pAvP ⊄ avP) then      ▷ check conflict
8:          return false
9:      else
10:         return
11:             isCandidate(node.parent, avP, prunelist)  ▷ navigate upwards
12:     end if
13: end function
```

Consequently, the two distinct values of the attribute will be immediate children of the root node. Hence, by directly navigating through the children of the root node, large parts of the tree can be instantly pruned, resulting in faster join execution.

During the creation of the strict order, the number of unique values of the attribute with the highest priority in the order influences the selection of the join algorithm. If the attribute appears in very few documents we will opt for FPTreeJoin_v1, otherwise, FPTreeJoin_v2 will be selected.

The **FPTreeJoin_v1** algorithm utilizes the benefits of the FP-tree to navigate immediately to branches that hold potential candidates for join, while, at the same time, ignoring branches that have conflicting elements. This algorithm consists of **two** parts. Using the helper table for navigating to branches with potentially joinable documents, in *Part* 1, the branch is validated by checking if it can fulfill the join condition. In *Part* 2, the nodes of the validated branch are traversed and the documents that fulfill the join test are produced. Additionally, in *Part* 2, all branches that hold attributes which conflict with the attributes of the considered document are added to a **prunelist**. The prunelist is used in *Part* 1 to condense the set of branches with potentially joinable documents. Thus, the branches in the prunelist that unquestionably store documents that cannot be joined, will be immediately ignored.

Algorithm 2 provides the details for *Part* 1 of FPTreeJoin_v1. As input, the method receives the attribute-value pairs *avP* for the investigated document, *node* an FP-tree node initially gathered through the header table and the list for branches to ignore called *prunelist*. This method traverses the tree upwards and determines whether the parents of the node are located on a branch that is in the prunelist or if they are labeled with an attribute-value pair that is in conflict with the attribute-value pairs of the analyzed document (Algorithm 2, Lines 6–7). If a parent node is labeled with a conflicting attribute-value pair, the method returns that the respective branch should be excluded from the search for join partners. By knowing how the FP-tree has been constructed, we

Algorithm 3 FPTreeJoin - Join computation

Require: Node *node* from the FP-tree, Set of attribute-value pairs *avP*, List of joinable documents *result*, List of branches to ignore *prunelist*

1: **function** JOINDOCUMENTS
2:     **for** *child* **in** *node.children* **do**
3:         $cAvP = child.avPair$
4:         **if** $cAvP.attr \subset avP.attrs$ **and** $cAvP \not\subset avP$ **then** ▷ conflict
5:             $prunelist.add(child.branchId)$ ▷ only for FPTreeJoin_v1
6:             **continue**
7:         **end if**
8:         $result.add(node.documents)$ ▷ joinable documents
9:         **joinDocuments**$(child, avP, result, prunelist)$
10:     **end for**
11:     **return** *result*
12: **end function**

are certain that if a parent node has a conflicting value then all the children will also have conflicting values. If there are no conflicting elements and the root node is reached, the branch can be further examined for identifying join partners (Algorithm 2, Line 2–3).

Part 2 of the FPTreeJoin_v1 algorithm is presented in Algorithm 3. The algorithm navigates through the branch provided as input in a top-down fashion deciding whether the documents, represented by the nodes of the branch, are join partners for the investigated document. Document ids stored in the nodes for which the join test is satisfied are collected (Algorithm 3, Line 8) and the algorithm continues to navigate down through the branch (Algorithm 3, Line 9). If the join test is unsatisfied, the *branchId* of the current child node is added to a prunelist and the algorithm disregards the successive children of the current node (Algorithm 3, Lines 4–7).

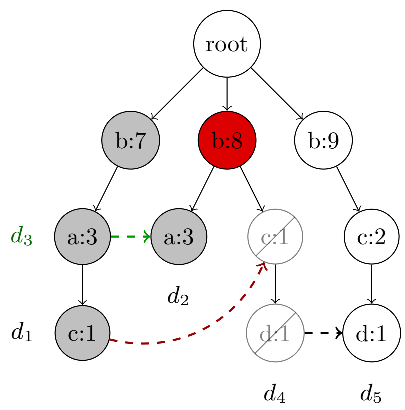


Figure 4.5: Example join for the FP-tree from Figure 4.4 using FPTreeJoin_v1 and joining document d_1.

Algorithm 4 FPTreeJoin_v2 - Part 1

Require: Document d_i, Root of FP-tree *node*, Number of attributes that appear in all documents *num*

1: $result = [\,]$ ▷ Final list of joinable documents
2: **while** $j < num$ **do** ▷ consider ubiquitous attributes
3: $\quad child = root.children.get(d_i.avPairs[j])$
4: $\quad node = child$
5: $\quad result.add(node.documents)$
6: **end while**
7:
8: $result.add($**joinDocuments**$(node, d_i.avPairs, result, [\,]))$
9: **return** *result*

As an example for the FPTreeJoin_v1 algorithm, consider Figure 4.5 where we search for the joinable documents of document d_1. The algorithm starts by checking the first attribute-value pair of document d_1 which in our example is $b : 7$. Consequently, Algorithm 2 is called by passing the first node labeled with $b : 7$. Because this node is an immediate child of the root, the respective branch is validated and can be traversed for detecting joinable partners. Thus, Algorithm 3 is called and by navigating through the children of node $b : 7$ it identifies document d_3 as a joinable document. The algorithm continues with the next attribute-value pair of document d_1. Since the first node labeled with $a : 3$ is located on an already covered branch, the algorithm follows the link to the second branch. Invoking Algorithm 2 leads to the check of the parent node labeled with $b : 8$. Because this node is in conflict with the attribute-value pair $b : 7$, the complete branch is pruned and the check for join partners is not performed. As last, the attribute-value pair $c : 1$ is checked. Since the next node labeled with $c : 1$ is located on a pruned branch, the algorithm informs that only document d_3 is a joinable partner for document d_1.

The **FPTreeJoin_v2** algorithm does not use the helper table for finding equally labeled nodes. Instead, it directly navigates through the children of the root node. The algorithm is built around the idea that there is an attribute that is present in all the documents. We further generalize this idea by considering all the attributes that appear in every document. Having n attributes of this kind and by knowing how the FP-tree is built, we can be certain that the first n levels of the tree will be made up of these attributes. Thus, for the first n levels, we can immediately gather the equally named nodes and ignore the rest of the nodes because we can be certain that they will be in conflict with the attribute-value pairs of the considered document. Consequently, Part 1 will differ for this algorithm as presented in Algorithm 4. Through the creation of the FP-tree, we can obtain information about the number of attributes that are present in all the documents and pass this information through the variable *num* to the algorithm. This is used in order to quickly navigate through the first levels of the FP-tree consisting of the ubiquitous attributes (Algorithm 4, Lines 2–6). All documents that have been encountered along the path are stored in the *result* list as resulting joinable documents for document d_i. Once a large portion of the FP-tree is pruned and there are no more attributes that are present everywhere, the method *joinDocuments* is called (Algorithm 4, Line

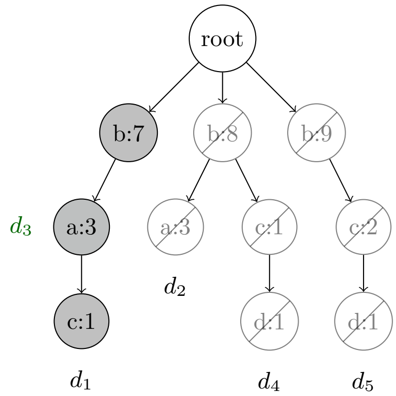


Figure 4.6: Example join for the FP-tree from Figure 4.4 using FPTreeJoin_v2 and joining document d_1.

8). Using the *joinDocuments* method the algorithm will continue to navigate toward the leaf nodes of the tree and gather all the documents that can be joined with the analyzed document. Unlike the previous algorithm, This version of the join algorithm does not need the prunelist (Algorithm 3, Line 5) because we are certain that the same branch will never be covered twice.

An important remark for the *joinDocuments* method (Algorithm 3) is that not all document ids from the nodes can be added as joinable documents. If there are no ubiquitous attributes in the documents, the loop in Algorithm 4 will not be executed. As a result, the currently investigated document will not have any in common attribute-value pairs with the branch of the node *node*. For that reason, it is important to keep track of all the shared attribute-value pairs of the investigated branch with the document d_i and update them for every new child node. A document id can be stored as a joinable document only if the number of shared attribute-value pairs is greater than zero.

As an example for the FPTreeJoin_v2 algorithm, consider Figure 4.6 where we search for the joinable documents of document d_1. During the computation of the strict order of the attribute-value pairs, we have identified that there is one attribute that appears in all the documents, attribute b. Thus, we are certain that the first level of the FP-tree will consist of only this attribute. Consequently, we take $b : 7$ as the first attribute-value pair of document d_1 and search for the equally named node in the first level of the FP-tree. Hence, we immediately prune the branches that start with the nodes $b : 8$ and $b : 9$. Having no more attributes that appear in every document, Algorithm 3 is invoked which by navigating through the children of the node $b : 7$ identifies document d_3 as a joinable document for the document d_1.

Depending on the data distribution, it might be beneficial to combine the two versions. This can be the case when some attributes appear in every document but once the levels containing them are covered, there is a large number of branches still to be examined. In this situation, the direct navigation through the children may lead to an increased execution time and thus, we may benefit from the first version that utilizes the pointers between the equally named nodes.

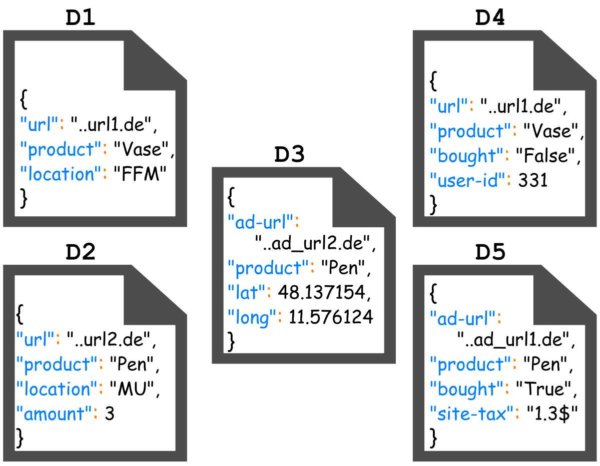


Figure 4.7: Example set of JSON documents generated from ad clicks.

4.5.3 Beyond Self Joins

Although the idea behind our proposed natural join algorithm for schema-free data is easily conveyed through the self-join paradigm, the join algorithms can be generalized to handle joins between two or more datasets. For instance, consider the example set of JSON documents presented in Figure 4.7. Assume that one dataset consists of all documents that were generated by following a link from an advertisement (*ad-url*), and the second dataset contains all other documents that were created by following any other type of link. Consequently, we would like to perform a natural join between these two datasets of JSON documents, which will be of benefit since we can extend the information about the documents from different platforms.

When considering joins between multiple datasets, the main challenge lies in identifying the dataset that needs to be indexed as an FP-tree, so that the time needed for computing the natural join is minimized. Below, we outline best practices for selecting the dataset that needs to be indexed.

Intuitively, the *size* of a dataset and the number of unique attribute-value pairs can be directly utilized for identifying if a dataset should be indexed. Having fewer documents and attribute-value pairs can directly lead to fewer branches in the FP-tree and, thus, a smaller FP-tree. Naturally, searching through a smaller FP-tree will lead to a lower join execution time and the indexed tree will occupy less memory. However, this is not a silver bullet. As an example, consider that both datasets have a similar number of documents and attribute-value pairs or that the majority of the indexed documents have a different schema. Then, relying solely on the size of the datasets can be misleading. Consequently, we may end up with a tree that cannot be efficiently traversed by our proposed join algorithms.

If it is not possible to choose which dataset needs to be indexed based on the size, we have identified that we should rely on the *number of ubiquitous attributes*, i.e., attributes that appear in all documents.

For the FPTreeJoin_v2 algorithm, the group with the highest number of ubiquitous attributes should always be indexed first. This conclusion is not surprising because the more attributes that are present in every document the faster the algorithm will navigate through the higher levels of the FP-tree. Therefore, even if the documents have mainly different schemas and the tree has many branches, the algorithm will immediately ignore all the branches that do not agree on the ubiquitous attributes and promptly return the join results.

For the FPTreeJoin_v1 algorithm, there will be no such attributes that appear in all the documents. Nevertheless, there can be attributes that appear in many documents although not in all of them. Thus, for this algorithm, we have identified that unlike for FPTreeJoin_v2, we should always index datasets that have fewer attributes that appear in many documents. This is not surprising because the more an attribute appears in the documents the higher the chance that the same value for this attribute is present in different documents. Having documents with different schemas would mean that these same attribute-value pairs will then be located on different branches in the corresponding FP-tree. Because the algorithm uses the helper table to navigate through branches with equally named nodes, it would need to check many branches which directly affects the execution time of the algorithm. Thus, if a dataset does not have many attributes that appear in most of the documents, the posting lists in the helper table will be smaller contributing to a faster execution time of the algorithm. The effects of the right indexing strategy for both approaches are shown in Section 4.7.6.

For joins between more than two datasets, we additionally need to consider the optimization of the join order. Since we have the necessary datasets already indexed, as a preliminary approach, we always join with the next most selective tree. We classify a tree as more selective if it has fewer frequent attributes or it has many ubiquitous attributes that have a high number of unique values for FPTreeJoin_v1 and FPTreeJoin_v2, respectively. Although this is a natural way to tackle the optimization problem, it can be far off from the optimal solution. Identifying the optimal join ordering is a research problem on its own and out of the scope of this thesis.

4.6 Practical Tweaks: Handling Dynamics and Countering Low Value Variety

4.6.1 Updating and Recreating Partitions

Over time, new documents with previously unseen attribute-value pairs will arrive in the system. It is the responsibility of the Assigners to handle these documents since they represent the components that connect the partitions with the Joiners.

We define the updating of the partitions as adding a single document to the already created partitions. The updating is triggered by the Assigners and it is performed in the Merger. Because of the strict requirement of having only one Merger, updating the partitions for every document with previously unseen attribute-value pairs is impractical, for two reasons. Firstly, the Merger will be congested with many messages, which will directly affect the performance of the whole system. Secondly, we risk incorporating extremely rare occurrences into the partitions that spoil the quality of the partitions in terms of replication. Thus, a document is a candidate for an update if it has an attribute-value pair that has appeared at least δ times. We consider that the attribute-value pairs that appear less than δ times are a result of a unique event and as such, they should not be considered for updating the partitions. However, we guarantee to produce all the joinable documents for a given window. Hence, documents that do not satisfy the update requirement are emitted to all Joiners.

As indicators for triggering the repartitioning, we consider the load balance and the replication. With the creation of the final partitions, the Merger computes the load balance and replication of documents that are a direct result of the computed partitions. When the Assigners detect that either the load balance or replication factor has increased over a prespecified threshold θ, they inform the Partition Creators and the Merger that in the next window recomputation of the partitions should be initiated.

4.6.2 Introducing Value Variety for Scaling

Attributes that have only a few unique values and are present in all of the documents will limit the scalability of the partitioning approach, as there will not be enough partitions created. As an extreme but not unrealistic case, consider documents containing a Boolean parameter, where by intuition, only two partitions should be created. In order to scale the computation to more than just two machines, there needs to be more variety in the domain of attribute values. The approach we conduct first searches for an attribute, referred to as *disabling attribute*, which appears in **all** the documents and has a number of unique values that is smaller than the required number of partitions. To enable the creation of the required number of partitions m without introducing unnecessary replication, the values of the disabling attribute will be "concatenated" with the values of the next attribute that appears in **most** of the documents and has the smallest number of unique values, called *combining attribute*. Since the number of artificially created values can still be smaller than the required number of partitions m, the proposed expansion will be repeated until the artificially created attribute has enough distinct values to satisfy the required number of partitions m.

As shown in Section 4.7, this expansion of attribute values is also a necessity for the disjoint sets algorithm [10], because, without it, it can practically never create enough partitions to populate the available number of machines.

Note that this expansion approach can introduce additional replication of documents since there can be documents in which the combining attributes do not appear—and hence, the artificial value cannot be created at all. As a consequence, to guarantee correct join results, such documents will be emitted to all machines, thus, creating additional network load (replication). The expected replication can be estimated by $p_{na} * m$, where p_{na} represents the percentage of documents where the parameter av_i does *not* appear and m is the number of partitions (i.e., machines).

4.7 Experiments

We have implemented our proposed topology using Java and Apache Storm 1.2.2. For carrying out the experiments, the topology was deployed on a cluster of 8 servers, with Linux (Ubuntu 14.04.5) as operating system. Every machine in the cluster is equipped with two 6-core Intel Xeon E5-2603 v4 CPUs @1.7 GHz and 128GB RAM.

4.7.1 Competitors

We investigate the performance of the following partitioning algorithms.

- Our Association Groups based partitioning approach (**AG**)
- The approach based on set covers (**SC**) [10], tuned for low communication overhead. The SC algorithm is based on the set cover problem and tries to find the smallest set of subsets through which the occurring sets can be represented. In this work, the attribute-value pairs of one document represent a set. It starts by creating initial partitions where in every iteration the set with most uncovered attribute-value pairs and minimal number of covered attribute-value pairs is selected. For assigning the remaining sets to the initial partitions, in every iteration, the set with least number and most uncovered attribute-value pairs is selected and assigned to the partition that has the least load and the most attribute-value pairs in common with the selected set.
- The disjoint sets (**DS**) partitioning approach [10]. The DS algorithm creates connected components called disjoint sets by combining all the sets that share at least one attribute-value pair. Every attribute-value pair belongs to one and only one disjoint set. The partitions are created by assigning every disjoint set to exactly one partition. By creating nonoverlapping partitions, the DS algorithm achieves perfect replication of having every document in exactly one partition.

For the FP-tree–based join algorithm, initially, we consider only FPTreeJoin_v2 in our streaming scenario since the considered datasets consist of Boolean attributes. Thus, we compare this version with two algorithms, the Nested Loop Join (**NLJ**) and Hash–Based Join (**HBJ**). The HBJ creates a hash table on the individual attribute-pairs as keys, such that documents that overlap in at least one attribute-value pair can be determined efficiently, essentially resulting in some sort of an inverted index over the contents of the documents.

For a detailed analysis, we compare our two versions of our FP-tree-based join algorithm to additional competitors that can store the data in memory. We perform this analysis to investigate further information together with the estimation time, such as the memory consumption and the effects of the indexing strategy on the execution time. As competitors, we used the following approaches.

- First, an Indexed Nested Loop Join (**INLJ**), that employs a hash index for one attribute that is used for reducing the number of comparisons. As an indexed attribute, the attribute with the largest number of distinct values is selected since it directly reduces the number of documents that need to be scanned. This is not a pure hash-based join algorithm since we do not create a hash index for every attribute in the documents.

- We further compare to **HSQLDB**[1], a relational in-memory database, where we create an index for the same attribute as in the INLJ algorithm. We have investigated the execution time of the join query by creating all possible combinations of indexes, however, a single index for the attribute with the most unique values resulted in the best execution time.
- Additionally, we employ **MySQL**[2], an open-source relational database. For a fair comparison, we create tables whose content is stored in memory rather than on disk. As for HSQLDB, we investigated the creation of all possible index combinations and we always report the result with the smallest execution time.

4.7.2 Datasets

The experiments in the streaming scenario were carried out using two different datasets. The real-world dataset (**rwData**) that consists of server logs of a mid-size company. The server logs, gathered from five servers, provide information about the user logins and file accesses. The data consists of 46 *million* JSON documents collected in a period of 105 *days*. To simulate a streaming environment, we take the daily produced amount as the number of documents produced every three minutes. We consider the whole data as a stream of incoming tuples, where the difference in the fields of the server logs fulfills our schema-free requirement for the documents. The sample in Figure 4.1 gives an impression of how the data looks.

For the synthetic dataset (**nbData**), we employ the NoBench JSON data generator proposed by Chasseur et al. [28]. The NoBench data generator creates an array of JSON objects with several attributes and in our work, every JSON object represents a separate JSON document. To create the possibility of having joinable documents, we remove the attribute *num* from the data generator because this attribute is unique in all the objects that are produced. It is important to note that the NoBench dataset consists of largely diverse elements, resulting in situations where every successive window contains documents with many previously absent attributes. Surprisingly, the same phenomenon holds for the real-world dataset. By investigating several window sizes, we deduced that in every subsequent window large number of the documents consist of previously unseen, new attribute-value pairs. This will be clearly visible in our experimental evaluation.

In addition to these two datasets, for evaluating the local join algorithms, we use a second real-world dataset (**apData**), with up to 50 different attributes. The apData dataset consists of flight information gathered from different sensors in Kaiserslautern, Germany. The documents in the dataset consist of information about the flight such as the longitude and latitude of the plane, the flight velocity, the unique identifier of the aircraft, the heading, and other information.

[1] http://hsqldb.org/
[2] https://www.mysql.com/products/community/

For the in-depth experiments concerning the local join algorithms, for all the above mentioned datasets, we use up to $500,000$ JSON documents.

4.7.3 Performance Metrics

During the execution of the aforementioned partitioning algorithms, we gather the following performance measurements.

- **Replication:** we measure the replication for every window as the average number of times that the same document has been sent from the Assigners to the Joiners. We also consider the documents that have unseen attribute-value pairs that do not satisfy the requirement for updating the partitions.
- **Maximal processing load:** we define the processing load of a single Joiner as the percentage of the assigned number of documents out of the total number of documents emitted for a given time window. The maximal processing load represents the highest processing load at one of the Joiners.
- **Load balance:** to assess the quality of the partitioning in terms of balanced load across the compute nodes, we measure the Gini coefficient [131]. It represents how much the actual load distribution deviates from a perfectly equal load distribution. A Gini coefficient of value 0 indicates that the elements are equally distributed among the machines, whereas a Gini coefficient with value 1 indicates the worst possible distribution of the documents across the machines.

When evaluating the local join algorithms, we report on the *execution time*, *memory consumption*, and *the effects of the indexing strategy on the execution time*.

4.7.4 Configuration Parameters

For testing the performance of our topology several settings for different parameters were examined.

Number of partitions m: the number of partitions is directly connected to the number of Joiners. There will be as many Joiner instances as the number of partitions. By increasing the number of partitions, the load for every instance of the Joiner is minimized but on the other hand, the replication can be increased by spreading the same documents on different machines. In our experiments, we performed measurements for $m = 5$, $m = 8$, $m = 10$, and $m = 20$.

Window size w: the size of the window specifies the number of documents that will be used in the Partition Creators for creating the initial partitions. Through the size of the window, the replication is directly affected. If the window is too small, the created partitions are not representing the data well, which leads to many documents having attributes that are not present in any of the m partitions. For our experiments, the size of the window was set to 3, 6, and 9 minutes.

Repartitioning threshold θ: the repartitioning threshold is a boundary that defines when a recomputation of the partitions should be initiated, as described in Section 4.6.1. When the replication or processing load has changed for more than the repartitioning threshold, the Assigners inform the Partition Creators that in the next window a recalculation of the partitions should be performed. This parameter directly influences the execution time of our approach. A smaller threshold indicates more frequent computation of partitions whereas a larger value indicates higher network traffic and a higher load per machine. We set the repartitioning threshold to 0.2 and 0.6, where a value of 0.2 means that the replication or processing load has increased by more than 20%.

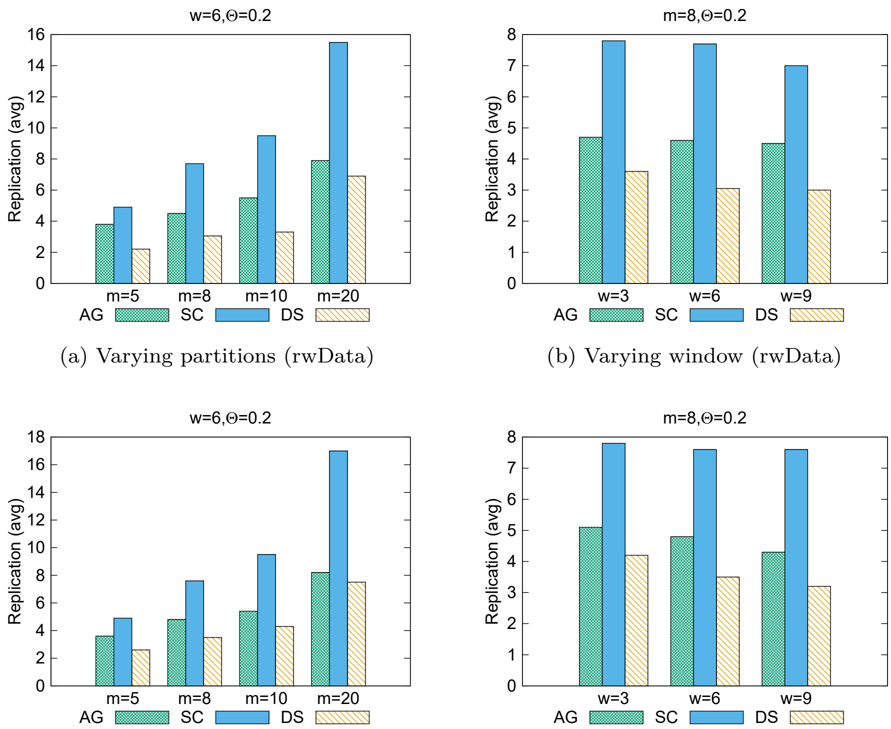
4.7.5 Evaluation of the Proposed Topology

In the following subsections, we investigate the effects of the configuration parameters on the proposed topology by varying one parameter while keeping the other fixed. If not specified otherwise, we use the default values for the parameters which are $m=8, \theta=0.2$, and $w=6$. All settings use six Assigners.

If a previously unseen attribute-value pair arrives at the Assigner at least $\delta = 3$ times (cf., Section 4.6.1), the partition recreation is initiated. We assume that an attribute-value appearing less than 3 times may be a result of a unique event and, as such, will not be considered as a valuable trigger. However, if a document contains an unseen attribute-value pair and the partitions are not yet to be updated, the document is emitted to all the Joiner instances, in order to preserve the correctness guarantees of the join result. The statistics for computing the quality of the partitions are performed at the end of every window. The Joiners compute and report the join results for every window. For the NoBench dataset, all partitioning approaches use the attribute expansion technique, presented in Section 4.6.2, as there is a Boolean attribute. Without the attribute expansion, the approaches will not scale to a larger number of partitions/machines. The disjoint sets algorithm (DS) still needs the expansion of attributes for the real-world data, because, no matter the size of the window, it constantly creates disjoint sets whose number is smaller than the number of partitions we aim at.

Replication

The changes in the replication concerning the different settings of the configuration parameters are presented in Figure 4.8. One can observe that the DS algorithm does not have the expected (perfect) replication of 1 (a document belongs to only one machine). This is a result of documents that do not match the existing partitions and, as such, are emitted to all machines. This fact contributes to the higher replication of the other algorithms, too. In Figure 4.8a and Figure 4.8c we can observe that the number of partitions has a large impact on the replication. The AG approach handles the increase in the number of partitions acceptably well, by creating partitions without many overlapping documents. Additionally, as the number of partitions increases, the comparison of the replication with respect to the worst-case scenario improves, indicating the scalability of our partitioning approach. On the other hand, the SC approach, no matter the number of partitions, always approaches the worst possible replication of sending every document to (almost) every machine. By increasing the window size, the AG and DS algorithms create more suitable partitions. Based on the measurements in Figure 4.8b and 4.8d, we see that this is not the case



(a) Varying partitions (rwData)

(b) Varying window (rwData)

(c) Varying partitions (nbData)

(d) Varying window (nbData)

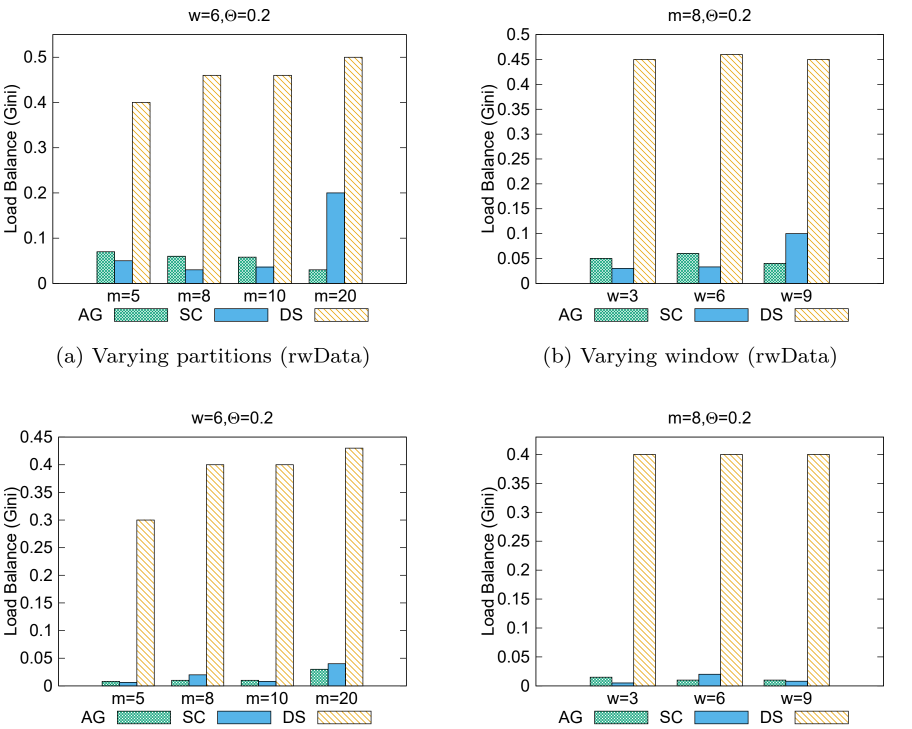
Figure 4.8: Replication of partitioning approaches in Storm topology.

for the SC approach. The SC algorithm, most of the time creates partitions where it needs to emit every document to almost all available machines.

For a distributed system the replication factor is of great importance, as it is directly related to the network traffic and also to the load the individual worker nodes have to cope with. From the results in Figure 4.8, we can conclude that the DS algorithm has the best replication followed very closely by our AG approach. The SC approach is completely not suitable for our documents and moreover, it only creates unnecessary overhead since the same results can be achieved by emitting every document to every machine. However, the large downside of the SC approach becomes evident once we inspect the load balance.

Load Balance and Maximal Processing Load

The processing load of a single Joiner is the percentage of the assigned number of documents out of the total number of documents emitted for a given time window. To measure the inequality of the processing load between the Joiners, we use the Gini coefficient. We use the measurements for the load balance and the maximal processing load to show how the compute nodes are balanced in terms of load and, whether the load balance is a result of an appropriate partitioning strategy or a result of replicating the same documents across the different machines.



(a) Varying partitions (rwData)

(b) Varying window (rwData)

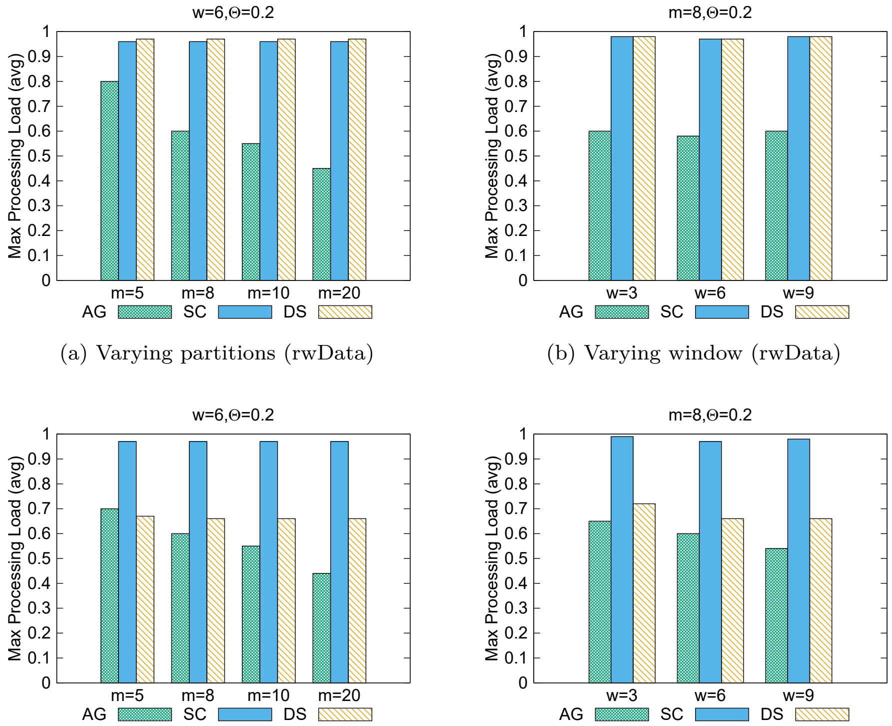
(c) Varying partitions (nbData)

(d) Varying window (nbData)

Figure 4.9: Load balance concerning partitioning approaches in Storm topology.

Figure 4.9 reports the load balance when changing the configuration parameters. As it can be observed, both the AG and SC approaches have satisfactory load balance, which for the synthetic dataset is not affected by varying the number of partitions or window sizes. For the real-world dataset, the load balance of the AG approach improves by increasing the number of partitions. This is a result of how the association groups are assigned to the partitions. By having more available partitions, the AG approach distributes the association groups more evenly across the partitions, creating a more balanced distribution of the documents. Unlike the AG and SC approach, the DS approach provides an inadequate distribution of the documents among the machines. This indicates that the DS algorithm creates disjoint sets which are extremely different in the number of documents that they have.

Based on the measurements for the processing load, we can conclude that both the AG and the SC approach are a good choice for partitioning schema-free documents. By examining Figure 4.10 we can see that this is not the case. The SC partitioning approach in every setting has at least one machine where almost the complete set of documents has been assigned. Thus, the balanced load is not a result of a good partitioning approach but it is achieved by replicating almost all the documents on the available machines. On the other hand, the AG algorithm has a lower maximal processing load when increasing the number



(a) Varying partitions (rwData)

(b) Varying window (rwData)

(c) Varying partitions (nbData)

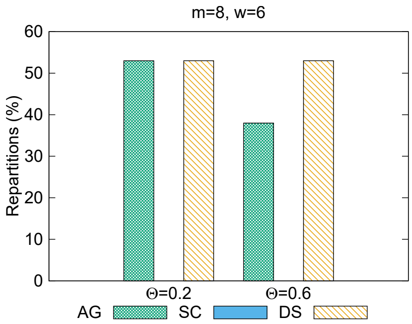
(d) Varying window (nbData)

Figure 4.10: Maximal processing load concerning partitioning approaches in Storm topology.

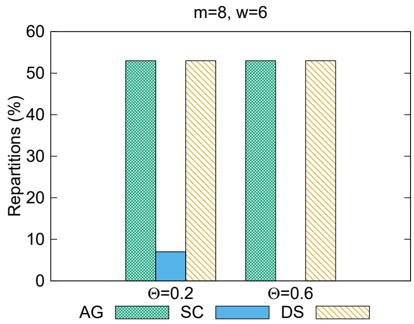
of partitions and the window size. This leads to the conclusion that the AG partitioning approach is highly scalable which is not a result of replicating more documents but of efficient computation of the partitions. Similar to the SC algorithm, the DS algorithm does not improve in terms of the maximal load when increasing the number of partitions. Moreover, for the real-world data, there is a single machine that receives almost all documents. This further underpins the expectation that the DS approach is not appropriate for larger JSON documents where a substantial number of documents are connected to one another via one or more attribute-value pairs.

Repartitions

The repartitioning is triggered either as a result of increased replication or unbalanced load. In Figure 4.11 we present the number of times the repartitioning was performed as a percentage of the total number of executions. The AG partitioning algorithm performs as expected for the real-world data, it has lower replication as the repartitioning threshold increases. For the NoBench data, the AG approach has the same replication rate for the different thresholds, 50% of the total number of elapsed windows. This is a result of the magnitude of unseen documents that arrive in every subsequent window and cause the algo-



(a) Varying threshold (rwData)



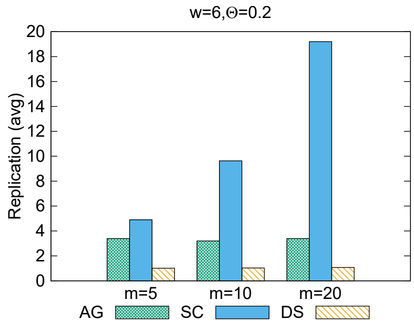
(b) Varying threshold (nbData)

Figure 4.11: Number of repartitions in Storm topology.

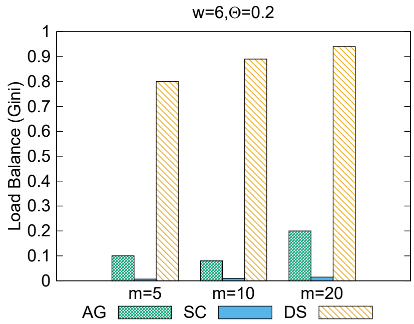
rithm to compute the partitions in every second window. The DS algorithm constantly has the same repartitioning rate. This comes as no surprise since having a previously unseen portion of documents in the subsequent window produces increased replication that undoubtedly will be higher than the originally computed average replication with value 1. The SC algorithm, from the first window, computes the worst possible partitions by replicating every document on almost every machine and from that point, there is no load balance or replication that is inadequate enough to cause the repeated computation of the partitions.

Ideal Execution

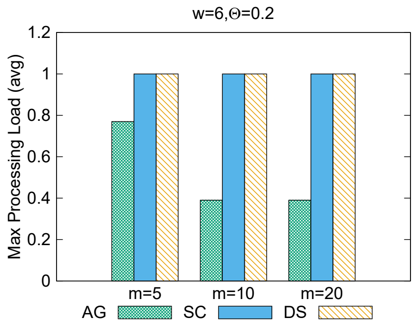
The presented results for replication, load balance, and maximal processing load are directly affected by the previously unseen attribute-value pairs that cause the containing document to be emitted to all the machines. Thus, the new attribute-value pairs directly increase the processing load and replication of documents. To demonstrate the execution of our approach when the data characteristics (in terms of co-occurring attribute-value pairs) are largely stable, we generated a dataset derived from the real-world data where we take one time-window and we repeat it multiple times. In every new window, we only add a predefined, small number of previously unseen documents. The results for the measurement are presented in Figure 4.12. Immediately noticeable is the improvement in the replication of the AG approach compared to the general case. Having a small number of previously unseen attribute values pairs indicates that the measured replication is a direct result of the partitioning algorithm. By increasing the number of available partitions, the AG approach continuously produces partitions with acceptable replication and improves the maximal processing load which contributes to better communication overhead and a more balanced load. The conclusions that we have made for the partitioning approaches are confirmed and even more evident. The results for the measuring of the maximal processing load show the scalability and efficiency of our approach and they prove that the excellent replication factor and load balance are indeed a consequence of a superior partitioning approach.



(a) Replication



(b) Load balance

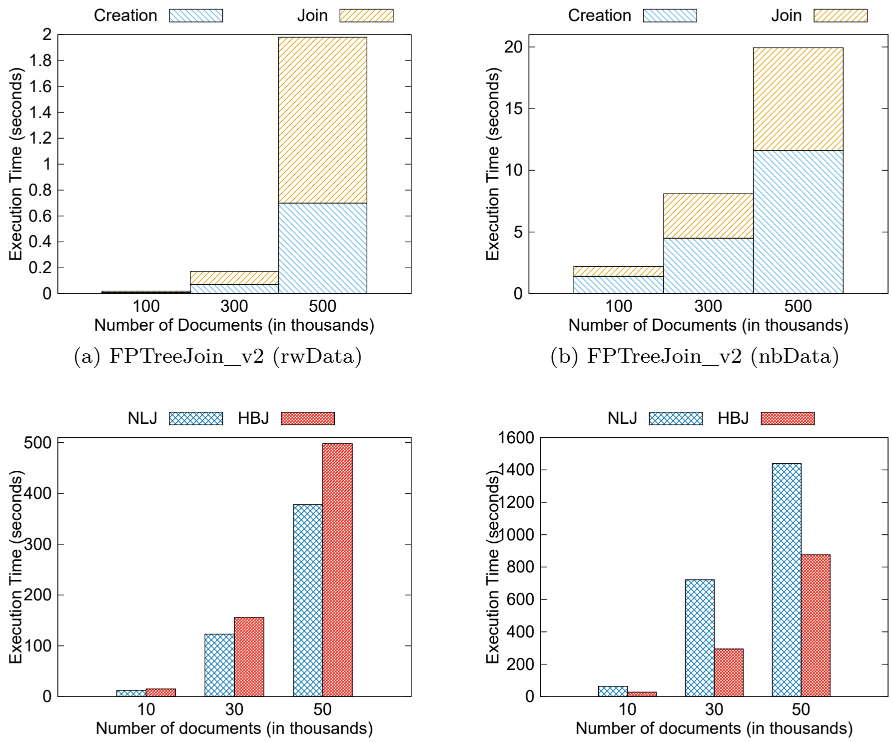


(c) Max processing load

Figure 4.12: Ideal execution concerning the considered partitioning approaches in an Apache Storm topology.

Join Execution Time in Streaming Topology

To the best of our knowledge, the existing approaches for joining documents in a streaming environment focus on performing equi or theta joins (or both) where the join attributes are known in advance or are of no concern. In our work, the attributes on which the join will be performed are unknown, but still, we want to exploit their importance when deciding if two documents are part of the natural join result. We compare our FPTreeJoin_v2 join approach with the natural join approaches used in traditional database systems, the Nested Loop Join (NLJ) and the Hash-Based Join (HBJ). We consider only the second variant since the datasets consist of Boolean attributes. The FPTreeJoin_v2 algorithm, as well as the baseline algorithms, are not distributed algorithms and, as such, are performed entirely within the Joiner instances. The execution time measurements of the join approaches for the real-world and synthetic datasets are shown in Figure 4.13. The x-axis in Figure 4.13 informs about the number of documents used for performing the join algorithm on a single node. The presented measurements depict the execution time of the join algorithms over a fixed number of documents on a single compute node and not a complete window in the streaming application. In our proposed topology, the join is performed for every window where every Joiner operates on the assigned documents. Thus,



(a) FPTreeJoin_v2 (rwData)

(b) FPTreeJoin_v2 (nbData)

(c) Competitors approaches (rwData)

(d) Competitors approaches (nbData)

Figure 4.13: Execution time of join algorithms in streaming topology.

every Joiner will execute the join algorithm for the assigned documents based on the created partitions.

Evidently, our proposed join algorithm is superior when performing joins over large, schema-free JSON documents. The time needed for both creating the FP-tree and performing the join approach when handling ten times more documents is orders of magnitude better. Furthermore, the execution time of the join algorithm is not significantly impacted by the data size. Even for the more diverse documents produced with the NoBench generator, the join is computed in a matter of seconds. One can immediately observe that the execution times of HBJ and NLJ become unacceptably high for real-time analysis. For joining $50,000$ documents, considered a small amount in a streaming environment, both approaches take several minutes. This becomes even more evident for the synthetic dataset where the NLJ algorithm needs more than 20 minutes and the HBJ algorithm around 15 minutes. One more interesting phenomenon that can be observed is that both algorithms perform differently for different datasets. For the real-world data, the NLJ algorithm outperforms the HBJ algorithm. This is a consequence of the high interconnection between documents, resulting in large document lists for a single hash value. For the synthetic dataset, the unique attribute-value pairs contribute to smaller bucket lists that drastically improve the execution time of HBJ.

Table 4.2: FPTreeJoin comparison (10, 000 documents).

Dataset	FPTreeJoin_v1		FPTreeJoin_v2	
	Creation	**Join**	**Creation**	**Join**
apData	0.15s	**0.09s**	0.14s	6.1s
rwData	0.07s	9.3s	0.07s	**0.05s**
nbData	0.29s	29.2s	0.28s	**0.1s**

4.7.6 Evaluation of Local Join Algorithms

In the following, we evaluate the two versions of the FP-tree-based join algorithms while comparing them to various in-memory competitors considering three different datasets. We report on the execution time and memory consumption of the approaches.

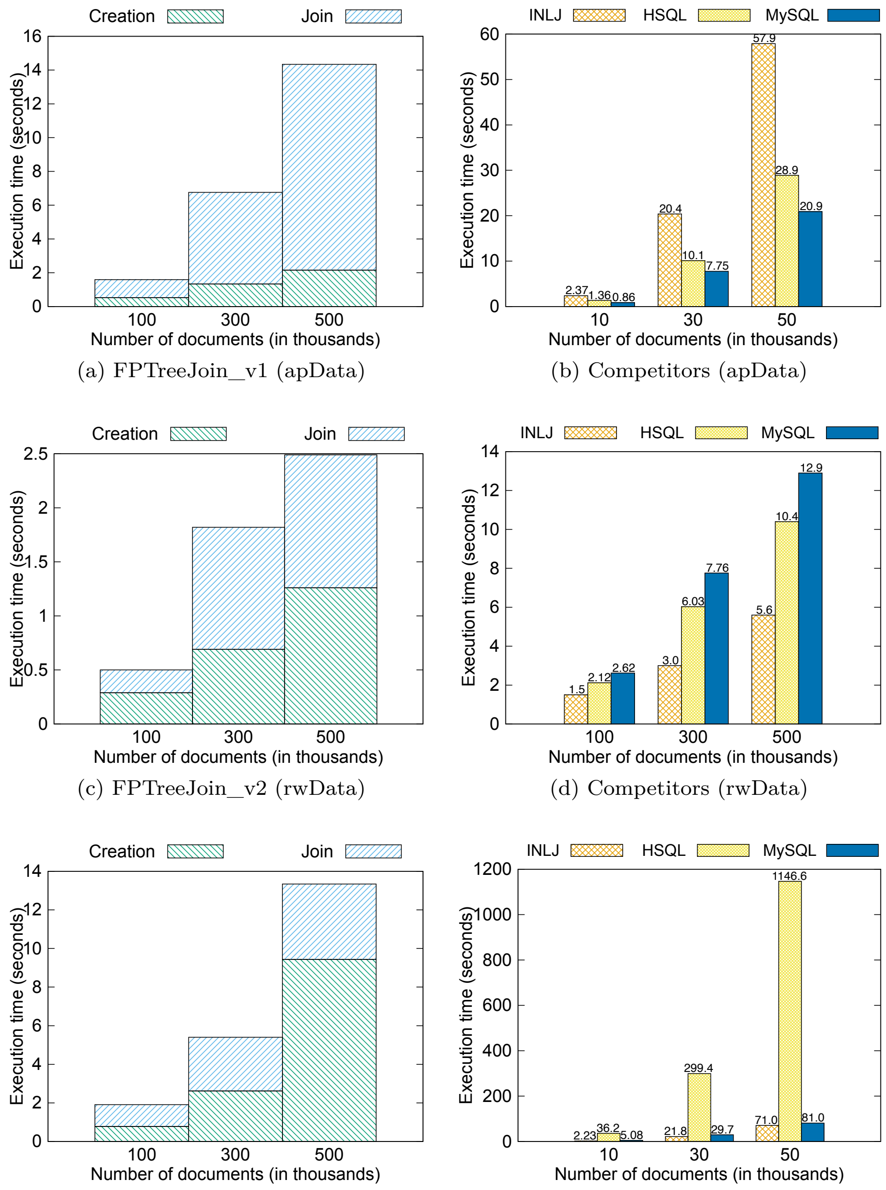
FPTreeJoin Analysis

Initially, we start by directly comparing the two versions of the FPTreeJoin algorithm for the considered datasets. The execution time, measured in seconds, of the two versions, when considering 10, 000 JSON documents, is presented in Table 4.2. We show both the time needed for creating the FP-tree and finding the join partners, where the highlighted cells indicate better time. When interpreting the presented results, it is obvious that the first version (*FPTreeJoin_v1*) is more suited for the apData dataset, where as the second version (*FPTreeJoin_v2*) is appropriate for the rwData and the nbData datasets. This is the case because both the rwData and nbData datasets have an attribute with only a few distinct values that appears in all the documents. Thus, by traversing through the direct children of the root, the FPTreeJoin_v2 algorithm can drastically reduce the number of comparisons. On the other hand, the documents from the apData dataset do not have such an attribute. Additionally, the present attribute-value pairs constantly appear in different combinations, resulting in an FP-tree with many branches. This comparison underpins the importance of having two flavors of FPTreeJoin for different data distributions. For the following experiments, when evaluating our approach to the competitors, we use the FPTreeJoin_v1 algorithm for the apData dataset and the FPTreeJoin_v2 algorithm for the rwData and nbData datasets.

Execution time

As next, we compare our algorithm to the considered competitors and we investigate the time needed for finding all the joinable partners when varying the throughput. Every experiment was repeated 5 times and we report the average execution time out of the 5 runs. The obtained results are depicted in Figure 4.14. The x-axis informs about the number of documents on which the join was performed and the y-axis reports the execution time of the algorithms.

When analyzing the presented results, it is apparent that the versions of our join algorithm drastically outperform the competing approaches for all the considered datasets. When examining the results for the apData dataset, for joining



(a) FPTreeJoin_v1 (apData)
(b) Competitors (apData)
(c) FPTreeJoin_v2 (rwData)
(d) Competitors (rwData)
(e) FPTreeJoin_v2 (nbData)
(f) Competitors (nbData)

Figure 4.14: Execution time comparison of local join algorithms (note the different scales on the x-axes for competitors vs. own approaches).

only 50, 000 documents, HSQLDB and MySQL need more than 20 seconds and INLJ needs almost one minute. On the other hand, for handling 500, 000 documents, our algorithm needs around 14 seconds. Although HSQLDB and MySQL use hash indexes which can help them for identifying potentially joinable documents, they still need to compare the values for the other (potentially not indexed) attributes. This is a direct reason for the increased execution time.

Table 4.3: Memory consumption of local join algorithms (Megabytes).

	# docs	**FPTreeJoin**	**INLJ**	**MySQL**
	10k	**11**	**11**	59
apData	30k	**31**	32.8	178
	50k	**49.2**	54.5	297
	100k	**26.4**	49	989
rwData	300k	**76.2**	146.3	2967
	500k	**134.9**	245.5	4945
	10k	54.2	**32.2**	168
nbData	30k	163.3	**97**	505
	50k	192	**120.7**	834

The difference in the execution time is even more evident for the nbData dataset. For $50,000$ documents, all competitors need more than a minute and HSQLDB even needs around 20 minutes for identifying the join partners. In contrast, our approach needs less than 14 seconds for handling 10 times more documents. Although the large number of attributes that are present in the documents do not affect our algorithm, they are a direct cause for the increased execution time of the competitors. The NoBench data generator can create more than 1000 different attributes [28]. Thus, all the competitors will have much more attributes, i.e., columns, with potentially empty values for comparison.

Finally, only for the rwData dataset, the competitors can be executed for the same number of documents as our algorithm. The reason for this is the *time* attribute that has a high number of unique values and is present in most of the documents, which combined with the small number of attributes and attribute-value pairs, contributes to the improved execution time of the competitors. However, still, for the same amount of documents, our approach always executes in at least two times less time.

Thus, by investigating datasets with a varying number of attributes and different distributions of the attribute-value pairs, when considering the execution time, it is evident that our proposed approach is superior to the competitors.

Memory consumption

We compare the memory consumption of our algorithm to the competitors and we present the results in Table 4.3. For the FPTreeJoin, we measure only the memory that is occupied by the FP-tree since the tree stores all the relevant information for the documents. We do not show any measurements about HSQLDB because the database management system does not expose any methods for measuring the size of the database. The only option was to measure the runtime memory of the Java Virtual Machine. However, the size of the database is not the sole contributor to the memory. Thus, to not provide misleading results, we proceed without the memory consumption results for HSQLDB.

As depicted, our algorithm has higher memory consumption than the competitors only for the nbData dataset. This is a direct consequence of the large number of different combinations of attribute-value pairs. For every document

Table 4.4: Execution Time per Indexing Strategy for FPTreeJoin_v1 (apData).

	Join Dataset (Many Freq. Attr.)	Join Dataset (Few Freq. Attr.)
Indexing Dataset (Many Freq. Attr.)	—	72ms
Indexing Dataset (Few Freq. Attr.)	53ms	—

Table 4.5: Execution Time per Indexing Strategy for FPTreeJoin_v2 (rwData).

	Join Dataset (1 Ubiq.)	Join Dataset (2 Ubiq.)	Join Dataset (3 Ubiq.)
Indexing Dataset (1 Ubiq.)	—	204ms	290ms
Indexing Dataset (2 Ubiq.)	182ms	—	184ms
Indexing Dataset (3 Ubiq.)	159ms	167ms	—

with a new combination of attribute-value pairs, there will be a separate branch created in the FP-tree. Consequently, instead of storing the same attribute-value pair only once, it will be duplicated in as many branches as there are documents with distinct combinations that include the attribute-value pair. On the other hand, because the hash index of INLJ is built on the attribute with the most unique values, naturally, will always lead to limited duplication of documents. The apData and rwData datasets do not have as many different combinations of attribute-value pairs as the nbData dataset. Hence, the succinct document representation in the FP-tree leads to reduced memory consumption of our algorithm when compared to the competitors. MySQL, due to the storage of numerous metadata information, as expected, has the highest memory consumption.

Effects of Indexing on Join Time

As next, we perform experiments that show that when performing a natural join between two datasets, the selection of the group of documents that will be indexed with an FP-tree (Section 4.5.3) has a direct impact on the execution time of the FPTreeJoin algorithms.

For the **FPTreeJoin_v1** algorithm we created two groups of documents of size of approximately $10,000$ from the apData dataset but with a different number of frequent attributes, i.e., attributes that appear in many documents. We measure the execution time of the FPTreeJoin_v1 algorithm for the two possible indexing options. The results are presented in Table 4.4. Evidently, the depicted results underpin the conclusions that were made in Section 4.5.3. If we index the dataset that has fewer frequent attributes (row 2, column 1) the algorithm achieves a better join execution time than when indexing the dataset with many frequent attributes. As already pointed out, the frequent attributes will contribute to the construction of FP-trees in which there will be more equally named nodes that are located on different branches. Thus, the helper table will have attribute-value pairs with larger posting lists and the time needed for following these navigational links is the main reason for the increased execution time.

For **FPTreeJoin_v2** we created groups of documents from the rwData dataset by varying the number of ubiquitous attributes, i.e., attributes that are present in all documents. We created three different groups of documents with one, two, and tree ubiquitous attributes, respectively. It is important to note that we did not consider a dataset without any ubiquitous attributes because then we would not be able to use the FPTreeJoin_v2 algorithm which is specifically designed around the presence of ubiquitous attributes. We measured the execution time when joining two groups for all of the group combinations and we depicted the results in Table 4.5. By analyzing the results, it is evident that the conclusions made in Section 4.5.3 are justified and directly influence the join execution time. It is clear that if we index the dataset with three ubiquitous attributes when joining with the dataset with one (159 ms) and two (167 ms) ubiquitous attributes we achieve better execution time than if we would index the dataset with one (290 ms) or with two (184 ms) ubiquitous attribute. Thus, whenever two groups of documents are joined, if the group with more ubiquitous attributes is indexed, the join algorithm achieves better execution time.

4.8 Summary

We addressed the problem of computing natural joins over schema-free JSON documents, which entails two core tasks: finding a feasible data partitioning schema and performing local query evaluation efficiently. For both aspects, we put forward novel solutions, harnessing core concepts of association analysis for partition creation and FP-tree–based join algorithms for local execution at the involved compute nodes. We further provided a countermeasure for cases of low attribute-value variety that inherently limit scalability. We reported on results and insights of a performance evaluation, which clearly showed the viability of the overall approach to handle large volumes of data in a resource-efficient manner. Further, performance gains of the partition approach over existing competitors were observed. In addition, the experimental results showed drastic performance gains of the FP-tree–based join algorithms over hash-based joins, nested loop joins, manually tuned index nested loop joins, and the realizations via HSQLDB and MySQL.

Chapter 5

Data Summarization in Resource Constrained Environments

5.1 Introduction

In the previous chapter, we addressed the computation of natural joins when working with schema-free data and considered a traditional streaming setting where the data is continuously generated from various applications or devices. What characterizes a traditional stream processing environment is that the processing of such a vast amount of data is still commonly done at centralized compute clusters where high computational power and network bandwidth are available for deep analytical tasks. The required data transfer from originating sources to such a centralized instance creates significant network traffic. This is especially visible when considering data sources on edge devices that collect and transmit data in real-time. Thus, in this chapter, we aim to minimize network traffic to gather data in centralized locations and, simultaneously, to speed up subsequent data processing. We propose to do so not by pushing the entire analytical processing to the edge devices —that often have limited compute power— but by compacting the stream using item-based summaries that represent the original data in an optimal way given an objective (utility) function. This chapter is based on our own publication at ADBIS 2022 [59].

Let us consider that we need to process a camera recording surveillance footage by checking the footage for outliers. When the camera records footage is in good quality, at a high frame rate, and there are multiple cameras at multiple locations, it is easy to imagine how the amount of data to be transferred and further processing become difficult to manage. A summary with only the most useful frames can reduce the amount of data while keeping the important information. The summarization could happen at a centralized location, but that means that all of the data needs to be sent through the network beforehand. This increases the network load and can result in higher latencies, as well as being cost-ineffective. Additionally, if the analysis needs to happen in real-time such as for theft detection and dispatching police services, we can immediately notice the disadvantages of such an approach. Thus, there is a clear benefit of performing the summarization at the edge devices, instead.

The most straightforward solution to summarize data are random sampling techniques [7, 157]. While they are understandable and easy to implement, they can result in important information being lost [31]. To overcome this problem, one solution is to use a **utility function**, which can measure the informativeness [20], representativeness [12,176], coverage [77], etc., of a selected data subset. With a utility function, the information gain for any new data point can be calculated and data points can accordingly be added to the summary. Although a utility function can be any function that assigns a non-negative score, typically, such functions belong to the class of *monotone submodular functions*, which means they are *non-decreasing*, and possess the *diminishing returns property*. Following Zhao et al. [172], we refer to these functions as *core item functions*, and the items identified by them are called *core items*.

As the problem of finding an optimal subset according to one of the functions is NP-hard [48], the main focus has been on finding good approximation algorithms [12, 23, 24, 84]. The related research on data summarization using submodular functions is vast. However, in data stream settings, the basis for most state-of-the-art approaches is the Sieve-Streaming algorithm, introduced by Badanidiyuru et al. [12], where data stream summarization is performed by maximizing a submodular set function subject to a cardinality constraint. However, when considering stream processing at the edge, Sieve-Streaming and the related variants Sieve-Streaming++ [84] and ThreeSieves [23] exhibit limitations. We develop our approach by considering the existing algorithms and addressing their limitations.

The related research investigating submodular function maximization typically focuses on text data [37, 102, 103] which enables the usage of natural coverage functions. However, such data is rarely encountered in edge applications. Data stream summarization has the the biggest impact on edge devices since it can greatly reduce network load. Additionally, the datasets are often not realistic representations of data generated on the edge but rather come from machine learning and bring diverse data points for classification purposes [12, 23, 24, 84]. Furthermore, most evaluations are performed on setups with high processing power, not representative of edge devices. This opens the opportunity to investigate the performance of existing algorithms in a realistic edge setting.

5.2 Contributions and Outline

In this chapter, we make the following contributions.

- We introduce an approach for fast core items computation in a data stream at the edge, which we call *SoftSieving*.
- We have tailored the state-of-the-art algorithms, Sieve-Streaming [12], Sieve-Streaming++ [84], and ThreeSieves [23] to be applicable inside an Apache Storm topology [2].
- We performed extensive experiments by using two real-world datasets measuring the processing times, latency, and utility values of the approaches.

The remainder of the chapter is organized as follows. Section 5.3 provides an overview of different utility functions that are applicable for core item sets computation. Section 5.4 reviews the shortcomings of existing approaches. Section 5.5 details our proposed approach for core items computation and the stream processing topology. Section 5.6 reports on the result of the experimental evaluation, and finally, Section 5.7 summarizes this chapter.

5.3 Utility Functions for Finding Core Item Sets

To summarize data streams, we focus on an approach of selecting a subset of items from the stream and forming item-based summaries. We consider utility functions which can measure the informativeness, representativeness, coverage, and other measurements, of the selected subset.

5.3.1 Utility Functions

To mathematically determine the quality of a summary, subsets of the data set need to be assigned a function value and then compared. This is the purpose of a utility function. For any new data item, the new value of the utility function can be used to determine whether to add it to the summary or not. A utility function f is any function $2^D \Rightarrow \mathbb{R}_{\geq 0}$ that assigns all subsets a nonnegative value [172]. The objective is to find the subset of size at most K with the best utility value. Formally put:

$$OPT = S^* = \underset{S \subseteq D, |S| \leq K}{\arg\max} \ f(S)$$

5.3.2 Submodular Functions

Suitable utility functions often belong to the class of *monotone submodular functions*. They have a diminishing returns property, i.e., $f(\{e\} \cup A) - f(A) \geq f(\{e\} \cup B) - f(B)$ for $A \subseteq B$. Additionally, they are monotone, so $f(\{e\} \cup A) - f(A) \geq 0$ for all e and A. The best approximation ratio for OPT in existing algorithms is $\mathcal{O}(\frac{1}{2} - \epsilon)$, which is also the theoretical maximum approximation ratio for the streaming setting [49].

Using the submodular functions as utility functions, we create data summaries, i.e., sets of core items. For a utility function f, the *marginal gain* for adding item e to the core item set S can be calculated as $\Delta_f(e \mid S) = f(S \cup \{e\}) - f(S)$. We assume $f(\emptyset) = 0$. In this work, we consider two different monotone submodular utility functions based on *active sets* and *exemplar-based clustering*.

Active set stems from *Gaussian Processes* (GP), which are used in nonparametric regression [137]. In GP the active set is used for efficiency and one way to choose an active set is the Informative Vector Machine (IVM) [94]. IVM is monotonic and submodular, as shown by Seeger [145].

The **k-medoids** problem aims to build clusters around exemplars from the set of data points [83]. To compute distances, it requires a nonnegative distance function. However, when working with data streams a problem arises since the distance to all data points needs to be known. Fortunately, the function is *additively decomposable* [116]. Thus, in our setting, we can continuously generate samples needed for the computation.

A detailed explanation of the active sets and K-medoids utility functions can be found in Section 2.3.

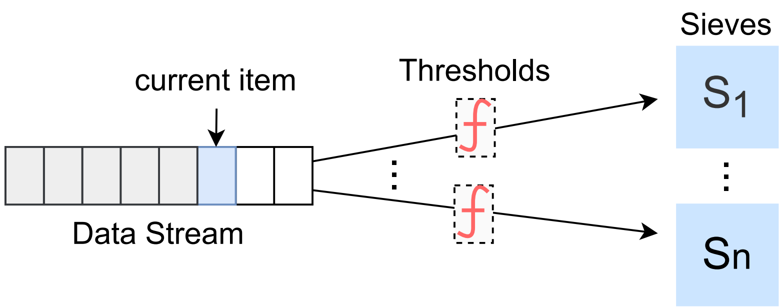


Figure 5.1: Computing sieves in a data stream.

5.4 Algorithms for Core Item Set Computation

In the following, we detail the existing approaches for data summarization focusing on their advantages and limitations. The Sieve-Streaming algorithm [12] and its variants, Sieve-Streaming++ [84] and ThreeSieves [23], provide solutions of high quality for data stream summarization. However, when considering data stream summarization on edge devices, we have identified that all of these approaches have certain limitations and can be further improved.

The *Sieve-Streaming* algorithm manages multiple sieves with different thresholds in parallel, as shown in Figure 5.1. For a new data item, if it satisfies the marginal gain, it will be added to any sieve for which it exceeds the specific threshold. In other words, each sieve filters items based on its threshold. At least one of the sieves is expected to have a fitting threshold and produce a good core item set. The elements from the sieve with the best utility value are returned as a result. The thresholds are approximated using the maximum singleton value $m_{sin} = \max_{e \in D} f(\{e\})$, i.e., the maximum value of the submodular utility function f for any single item e. The lower bound is m_{sin} since that value has already been reached. If the summary has size K, the best case would be K items increasing the function value by m_{sin}, resulting in upper bound of $K \cdot m_{sin}$. The number of sieves depends on the parameter $\epsilon > 0$, which also influences the quality of the result.

If Sieve-Streaming is implemented in an Apache Storm topology without modifications, no parallelism is possible since the best utility value over all sieves is needed for the output. This will lead to increased processing load and increased latencies. To overcome this, we split the algorithm into two parts. In the first part, sieves are processing tuples and finding core items. Then, all of these sieve bolts can send their core items to a single collector bolt, which updates the core item set with the best utility after each message.

When analyzing the Sieve-Streaming algorithm and topology, several limitations are immediately observable. First, the maximum singleton value m_{sin} can be updated for every new entry, which also means that the number of sieves can change when this update occurs. Hence, in the proposed topology, there will be additional communication overhead between the bolts responsible for computing sieves and the collector bolt, especially when the maximum singleton value m_{sin} changes frequently. Furthermore, although the algorithm achieves significant

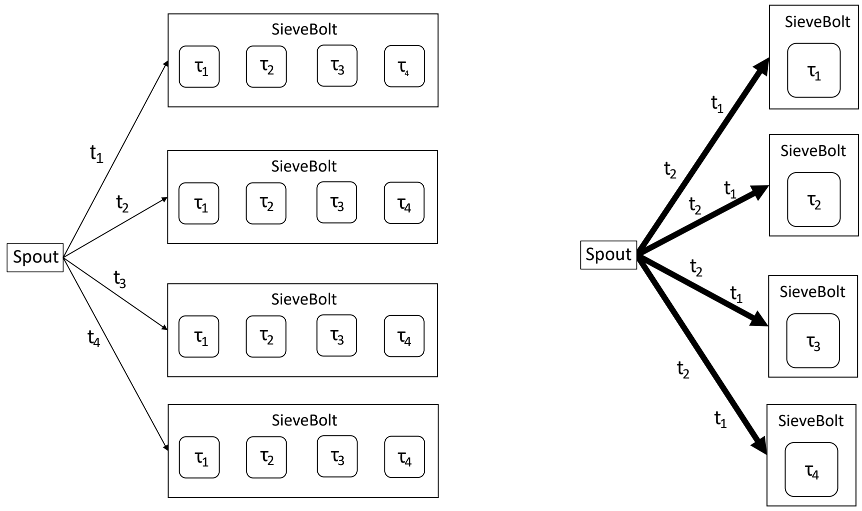


Figure 5.2: Possible directions for parallelization. Parallelization by splitting the stream (left) or splitting the sieves (right).

improvement over simple greedy algorithms such as reservoir sampling, the large number of created sieves can directly lead to problems with both storage and computation times.

Although *SieveStreaming++* [84] improves this approach by modifying the lower bound m_{sin} for the optimal solution OPT as the algorithm is running, still the same limitations are present. The number of sieves remains a problem for low ϵ. Additionally, since the Storm topology remains the same, the problem of additional communication overhead when m_{sin} changes frequently persists.

Unlike the previous approaches, *ThreeSieves* [23] maintains only one single sieve and decreases its threshold over time. The rules for adding an item to a sieve are the same as in Sieve-Streaming, however, when an item e is not added, a counter t is incremented by one. If the counter reaches the value of parameter T, the threshold value is decreased to the next-biggest estimate. This results in a threshold that is lowered more and more over time. To implement the ThreeSieves algorithm in a Storm topology we only need a single bolt. Although this eliminates the unnecessary communication overhead, it leads to an obvious limitation, i.e., lack of parallelization. This single bolt is bottleneck and if the stream arrives in high volume and velocity the processing will result in increased latencies. Additionally, the algorithm performance depends heavily on the choice of parameter T. If it is too small, the threshold will get too low and the summary will be filled quickly, resulting in lower utility. If it is too large, useful items can be missed.

5.5 SoftSieving

Considering the limitations of the existing approaches, we build our new approach called *SoftSieving*. The SoftSieving approach is developed by keeping the following design goals in mind.

- Use fewer sieves while keeping a high solution quality.
- Facilitate parallelization as much as possible. There should be sieve bolts that act independent of each other, and a global collector bolt.
- Make the processing as fast as possible by apt assignment of computations. Move expensive calculations to the collector bolt, to keep computation inside the sieve bolts, which process all data items, as fast as possible.

The first design goal steers in the direction of ThreeSieves [23] since its dynamic sieve can achieve high utility for many settings. The question then becomes how to increase the number of sieves. There should be bolt instances running in parallel that are responsible for computing their own sieves. Each instance should be independent of the others to minimize the communication overhead. Thus, as depicted in Figure 5.2, there are two general methods that we can take.

1. Split the stream over n bolts and have all of them use the same threshold(s).
2. The n bolt instances process all tuples, each with different threshold(s).

Since one of our design goals is to minimize the number of sieves, it is natural to consider the first method. However, we cannot expect much gain from having additional sieves in every bolt. This comes directly from the ThreeSieves algorithm since the solution quality is already high while using only one sieve. The sieve is built starting from the highest possible threshold and lowering the threshold over time. As a result, high-utility items at the beginning of the data stream can be wrongly discarded. To better include these items, we can employ the reverse strategy by starting at the lowest threshold and increase it. In this way, when the core item set of the sieve is filled with low utility items, it will be difficult to replace them later with higher utility items. Thus, we will include sieves that start from lower thresholds but still decrease them. The number of sieves depends on how much the thresholding calculations throttle the topology. Considering the first design goal, we will prefer a small, fixed number of sieves. Fixing the number of sieves prevents a rapidly increasing number of sieves and will enable fast inner-bolt processing. Since the data stream is split, we would require a collector bolt responsible for collecting the core items from the sieve bolts and building the core item sets. However, when adding a new item, every sieve bolt would need the current set of core items for their sieves at all times. Thus, there is a need for additional method to avoid these extensive updates.

To avoid numerous updates, we propose the usage of preemption [22]. Preemption means that once an item is added to the summary, it does not necessarily stay there but instead can be replaced by newer items at any point in time. The first K items are always added to the summary. Further items are swapped if they improve the solution by more than a defined threshold. If the summary is not fixed, we do not need to update the sieve bolts. They can continuously send the core items to the collector, which then checks if they improve the global solution when using swapping. The collector does not need to process as many items as the sieve bolts but still checking for every item in the summary quickly becomes infeasible with increasing K. Thus, the marginal gain of every item in the summary is stored. If the summary is full, new items are compared only to the item with the lowest gain. Consequently, we avoid the updates and ful-

Algorithm 5 SoftSieving

1: **for** $i = 1, ..., n$ **do** ▷ initialization
2: $\quad S_i \leftarrow \emptyset$, $t_i \leftarrow 0$, $b_i \leftarrow 0$
3: **end for**
4: **while** $(e \leftarrow D.next())\ != \ null$ **do** ▷ process items of stream
5: $\quad m_{sin} \leftarrow \max(m_{sin}, f(\{e\}))$, $O \leftarrow \{(1+\epsilon)^i \mid m_{sin} \leq (1+\epsilon)^i \leq K \cdot m_{sin}\}$
6: $\quad V_n \leftarrow n$ equidistant samples from O, and $\max(O)$
7: $\quad$ Delete all S_i such that $v_i \notin O$ **and**
8: $\quad$ Create new S_i all new $v_i \in O$
9: $\quad$ **for** $(S_i, v_i) \mid i = 1, \ldots, n$ and $v_i = V_n[i]$ **do**
10: $\quad\quad$ **if** $\Delta_f(e \mid S_i) \geq \frac{\frac{v_i}{2} - f(S_i)}{K - |S_i|}$ **then**
11: $\quad\quad\quad$ **if** $|S_i| < K$ **then** ▷ add item to sieve
12: $\quad\quad\quad\quad S_i \leftarrow S_i \cup \{e\}$
13: $\quad\quad\quad$ **else if** $b_i < K$ **then** ▷ sieve is full compare next K items
14: $\quad\quad\quad\quad e_s \leftarrow \arg\min_{e \in S_i} \Delta_f(e \mid S_i\{e\})$
15: $\quad\quad\quad\quad$ **if** $\Delta_f(e \mid S_i \backslash \{e_s\}) > \Delta_f(S_i)$ **then** ▷ is utility better?
16: $\quad\quad\quad\quad\quad S_i \leftarrow S_i \cup \{e\} \backslash \{e_s\}$
17: $\quad\quad\quad\quad\quad b_i \leftarrow b_i + 1$
18: $\quad\quad\quad\quad$ **end if**
19: $\quad\quad\quad$ **end if**
20: $\quad\quad\quad t_i \leftarrow 0$
21: $\quad\quad$ **else**
22: $\quad\quad\quad t_i \leftarrow t_i + 1$
23: $\quad\quad\quad$ **if** $t_i \geq T$ **then** ▷ if true, decrease sieve threshold
24: $\quad\quad\quad\quad v_i \leftarrow$ next lowest threshold from O
25: $\quad\quad\quad\quad O \leftarrow O \backslash v_{next}$, $t_i \leftarrow 0$
26: $\quad\quad\quad$ **end if**
27: $\quad\quad$ **end if**
28: $\quad$ **end for**
29: **end while**
30: **return** $\arg\max_{S_i \mid i=1,\ldots,n} f(S)$

fill our third design goal. To avoid excessive load in the collector and longer computations in the sieve bolts, the summaries will be restricted to size K.

Following these design decisions, as next, we present the **SoftSieving algorithm** as shown in Algorithm 5. In Lines 1–3, we initialize the algorithm. O is the set of thresholds, satisfying the defined lower and upper bounds. Compared to existing algorithms, not all thresholds are chosen from the set O, nor the maximum is only used. Instead, V_n gets n equidistant samples from O (Line 6). For the items from the data stream, we repeat steps 4 to 29. Similarly to the related approaches, we update m_{sin} and the sieves (Lines 5–8). Next, we add items to the sieves (Lines 10–20) such that an item will be added if Lines 10 and 11 are satisfied. The amount by which adding e to S_i increases the utility function must be big enough such that sieve S_i can still reach the approximated optimal value v_i, i.e., increase by $v_i - f(S_i)$ after adding $K - |S_i|$ items to the summary. To account for some items in the summary influencing the utility value more that others, $\frac{v_i}{2}$ is used instead of v_i. If a sieve is full, we compare the

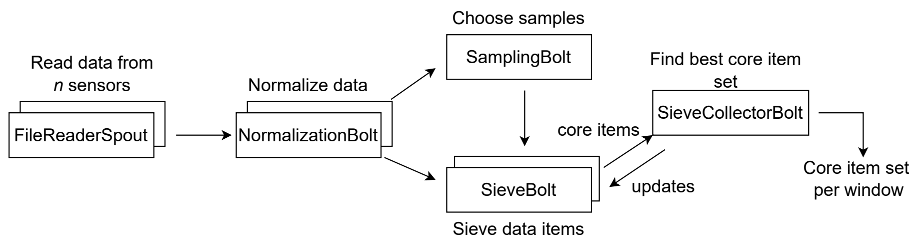


Figure 5.3: Apache Storm topology for SoftSieving approach.

next K items above the threshold against e_s, the item with the lowest utility gain from the core item set S_i (Lines 14–15). If item e offers higher utility, e and e_s are swapped. If an item is not added and Line 23 is satisfied, the sieve-thresholds are decreased (Lines 21–26). Finally, the sieve with the highest utility is returned (Line 30).

The actual Apache Storm **topology** responsible for realizing the SoftSieving algorithm is depicted in Figure 5.3. The topology consists of five main components. First, the multiple instances of the *FileReaderSpout* are responsible for emitting the data from the data stream. Since for some data streams the data needs to be normalized, we introduce the *NormalizationBolt*. There can also be several instances of this bolt. As explained in Section 5.3, we might need samples for computing the utility functions. For that reason, we introduce the *SamplingBolt*. The *SieveBolt* is responsible for maintaining and updating the sieves. The *SieveCollectorBolt* receives the core items from the *SieveBolts* and updates the core item set with the best utility. Although there can be several *SieveBolts*, there can be only one instance of the *SieveCollectorBolt*.

5.5.1 Theoretical Analysis

We will now analyze the time and space cost as well as the utility for SoftSieving, with a focus on the Storm implementation. For m bolt instances and n sieves per bolt, SoftSieving stores $\mathcal{O}(n \cdot (m+1) \cdot K)$ items. Each bolt instance has its own n summaries with up to K items.

For the time analysis, consider a single item e. For the bolt it is assigned to, it takes n evaluations of the utility function, which take time T_K for a summary of size K. If instead the core item set and an additional K items have been sent, no further evaluations are performed. In the collector bolt, there are two possibilities. If the global summary S for item e is not full yet, e is added to S in $\mathcal{O}(1)$ and the new utility value is calculated, taking T_K time. Otherwise, a swap is considered, which takes T_K time for the utility evaluation of $S \backslash \{e_{min}\} \cup \{e\}$. Overall, the collector receives a maximum of $2K$ core items from each summary, i.e., a total of $m \cdot n \cdot 2K$. For a data stream of size N split between m bolt tasks, the time complexity is $\mathcal{O}(\frac{N}{m} \cdot (n \cdot T_K) + 2nmK \cdot T_K)$. Considering tumbling windows of size w, we get $\mathcal{O}(\frac{N}{m} \cdot (n \cdot T_K) + 2nwmK \cdot T_K)$. When considering the updates of the maximum singleton value, since there is no upper bound on the number of updates, this increases to $\mathcal{O}(\frac{N}{m} \cdot (n \cdot T_K) + N \cdot T_K)$, or since $m, n << N$, $\mathcal{O}(N \cdot T_K)$. In practice, there will be significantly faster runtimes since the updates will not happen for every data item. For other approaches, the number of queries per element is often used [12, 23, 84], which is $\mathcal{O}(n)$ here since we always use n sieves.

Regarding the quality, we can consider the result in ThreeSieves [23] as a lower bound since we always include the sieve with threshold $K \cdot m_{sin}$. The authors prove that the solution S achieves an approximation to the optimal value OPT of $(1-\epsilon)(1-\frac{1}{\exp(1)})OPT$ with probability $(1-\alpha)^K$. When using n sieves, we expect to reach the optimal threshold v^* for one of them after $\frac{|O|}{2n} \cdot T$ items, instead of $\frac{|O|}{2} \cdot T$ for one sieve. This follows from choosing the thresholds equidistant from O, thus partition O into partitions of size $\frac{|O|}{n}$. In their proof, the probability of $(1-\alpha)^K$ comes from $P(v_1 = v_1^*, ..., v_K = v_K^*) > (1 - \frac{-\ln(\alpha)}{T})^K$, where, v_i^* are the thresholds of the sieve and $v_i = \Delta(S \mid e_i)$ are the marginal gains achieved by the greedy algorithm (for details see [23], Appendix). For SoftSieving, any of the n sieves can achieve these values, increasing the probability. We can approximate it with $1-((1-\alpha)^K)^n$ for n sieves, but recall that the n thresholds are not chosen at random, but to be equidistant. The swapping yields no such improved theoretical guarantees since we try to swap with the item with the lowest marginal gain. This is done for performance reasons but can result in swapping new item e only with its most similar item from the core item set, bringing minimal improvement.

5.6 Experimental Evaluation

The proposed topology and the considered competitors are implemented in EdgeWise [52]. The experiments were performed on a Raspberry Pi 4 with a 1.5 Ghz Quad-Core-processor and 8 GB of main memory, meant to simulate a realistic edge device. The Raspberry Pi runs an Apache Storm Cluster and ZooKeeper, where the topologies were submitted. We carried out experiments with the goal of answering the following questions.

1. Can we find summaries in a reasonable timeframe which is preferable to sending all data items to the core without summarization?

2. How well can the approaches handle increasing load from the IoT-device(s)?

3. Does the SoftSieving approach offer significantly better processing times or solution quality compared to existing algorithms?

4. How do the results vary for different summary sizes, window sizes, and ϵ?

Thus, we measure the total latency over the topologies (questions 1 and 3). We measure the processing times in the SieveBolts (questions 2 and 3) and the utility for all approaches (question 3). To address question 4, we perform experiments with varying summary sizes $K \in \{5, 20, 100\}$, window sizes $w \in \{1000, 10000, 100000\}$, and the parameter impacting the number of sieves $\epsilon \in \{0.1, 0.01, 0.001\}$. For processing times and total latency, the average over all tuples is taken, the utility is the average value over all windows.

5.6.1 Competitors

We considered the Sieve-Streaming [12], Sieve-Streaming++ [84], ThreeSieves [23], and Reservoir Sampling [157]. All the competitors were implemented in an Apache Storm topology. All the sieve-based algorithms are executed with parameters $K = 20, w = 10000, \epsilon = 0.1$, unless otherwise specified. ThreeSieves and SoftSieving additionally use $T = 100$ as a standard parameter and SoftSieving uses $n = 4$. The utility functions used are Active Sets and k-medoids (Section 5.3). The topologies are configured to have 10 SieveBolts and one SieveCollectorBolt. The NormalizationBolts have five instances. The SamplingBolt uses one instance.

5.6.2 Datasets

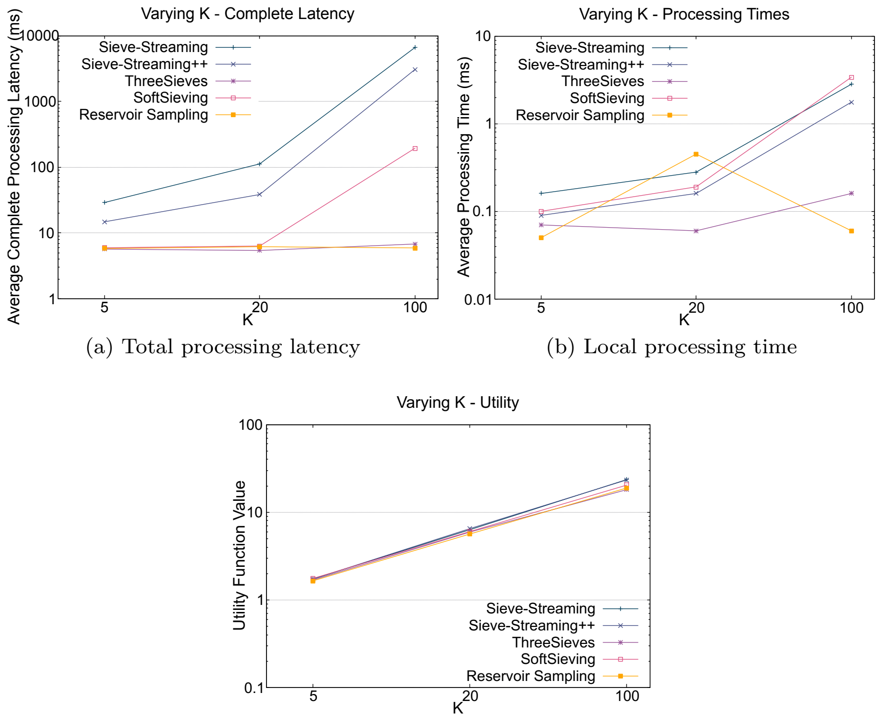
The approaches were evaluated on two real-world datasets, specifically chosen to simulate a real IoT scenario. The first dataset is a *telemetry dataset* from Stafford [148]. The data is collected from three IoT-devices, reading environmental sensor data in regular intervals. The measurements include temperature, humidity, CO, liquid petroleum gas, smoke, light, and motion, having 405184 entries. The used utility functions require a distance function, thus the distance between two items was implemented as sum of the euclidean distances of all measurements. The data is normalized in the NormalizationBolt to ensure even weight between all measurements. The second *images dataset* is an RGB-D dataset [108, 147]. It contains image frames from three Kinect sensors in an university hall. It includes both RGB and depth images, but we consider only the RGB frames. The images in the dataset are stored as 8 bits, 3 channels PPM images with 640x480 pixels. We preprocess the data and transform it into feature vectors of size 804. Distances are calculated as sum of the euclidean distances of all feature vector entries. Since the feature vector is normalized the NormalizationBolt is not necessary.

5.6.3 Experimental Results

First, we performed experiments for analyzing the total processing latency, local processing time and the active set utility function value when varying the size of the summary, the number of sieves, and the window size. When one parameter is varied, the other remain on their standard values.

Varying Summary Size (K)

Considering the total latency (Figure 5.4a), all approaches are affected by the summary size, but Sieve-Streaming and Sieve-Streaming++ perform the worst, with SoftSieving being up to an order of magnitude below them. All approaches have increased latency for larger K since it both takes longer for the summaries to be completed, and each utility function calculation over a larger set takes longer as well. Reservoir Sampling stays consistent since the sampling calculations are not affected by K. ThreeSieves is also barely affected, which means that the increase in computation cost is not notable when using only one sieve. The processing time of the SieveBolt (Figure 5.4b) increases as well. Reservoir Sampling, as it is not affected by the summary size, does not show a correlation to K. Figure 5.4c depicts the average utilities where a higher value



(a) Total processing latency

(b) Local processing time

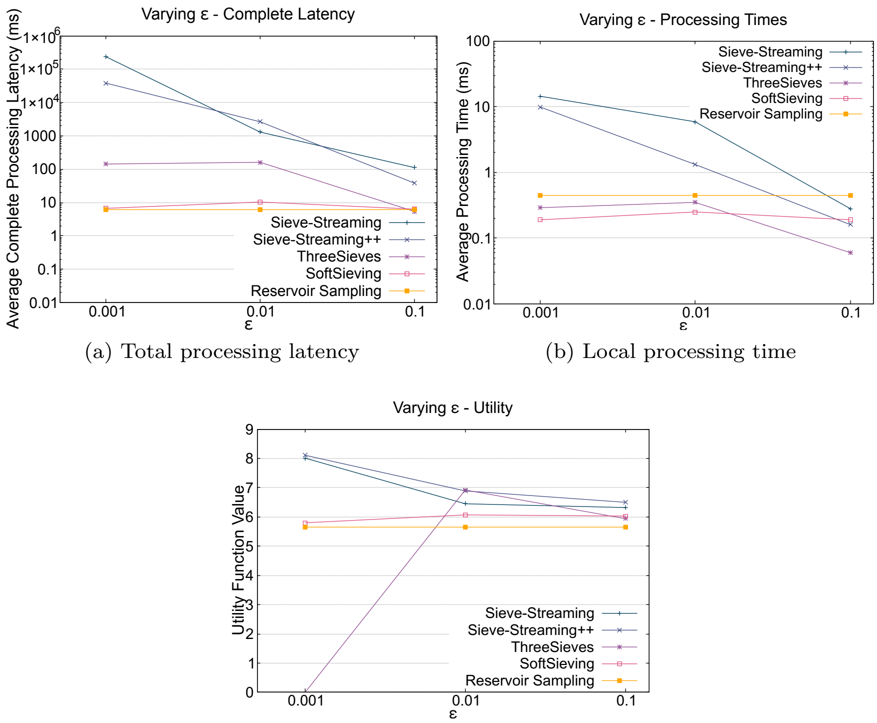
(c) Utility function value

Figure 5.4: Varying summary size ($K = 5$, $K = 20$, $K = 100$) - log scaled.

corresponds to a better approach. Clearly the results differ more for larger K. Sieve-Streaming++ achieves the highest utility. SoftSieving lies between Sieve-Streaming and ThreeSieves, but it constantly outperforms ThreeSieves. As expected, increasing the summary size leads to increased latency and processing time, while increasing the utility. Sieve-Streaming and Sieve-Streaming++ are the most dependent on K having latencies orders of magnitude larger compared to SoftSieving. SoftSieving also sees an increase in processing times, but is less affected in latency than Sieve-Streaming and Sieve-Streaming++. ThreeSieves and Reservoir Sampling are barely affected by K and produce the best latencies. However, they produce the lowest utility.

Varying Number of Sieves (ϵ)

Next, we will evaluate the performance of the algorithms when varying ϵ. As shown in Figure 5.5a, by increasing ϵ, the number of sieves decreases and with that the latency for all sieve-based approaches. This is most notable for Sieve-Streaming and Sieve-Streaming++, which, for the smallest ϵ, have a latency several orders of magnitude higher than the competitors. SoftSieving achieves lower total latency than ThreeSieves, which is a direct consequence of the improved parallelization. The local processing times follow the same trend (Figure 5.5b). SieveStreaming and SieveStreaming++ have high processing times for low ϵ. For ThreeSieves, the local processing time is lower for higher ϵ. The



(a) Total processing latency

(b) Local processing time

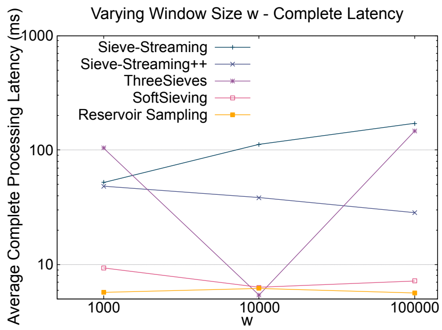
(c) Utility function value

Figure 5.5: Varying number of sieves ($\epsilon = 0.001$, $\epsilon = 0.01$, $\epsilon = 0.1$). Figure 5.5a and Figure 5.5b are log-scaled.

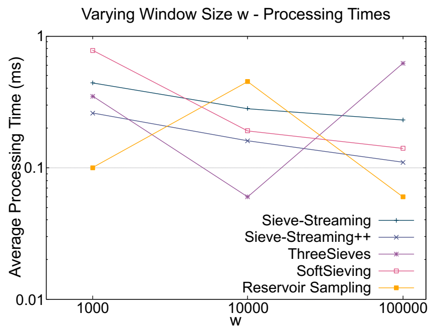
utility values in Figure 5.5c show the dependence of Sieve-Streaming and Sieve-Streaming++ on the number of sieves. Their utility decreases when ϵ increases and fewer sieves are used. SoftSieving shows a slight increase with increasing ϵ, while ThreeSieves has lower utility for both $\epsilon = 0.001$ and $\epsilon = 0.1$. The number of sieves does not change for ThreeSieves, but a lower ϵ means that, with constant T, it takes longer to decrease the threshold to a suitable value. The extreme case is shown for $\epsilon = 0.001$, where no items were added to the core item set since the threshold remained too high. Overall, there is a decrease in the latency and processing time for all approaches with increasing ϵ, except Reservoir Sampling.

Varying Window Size (w)

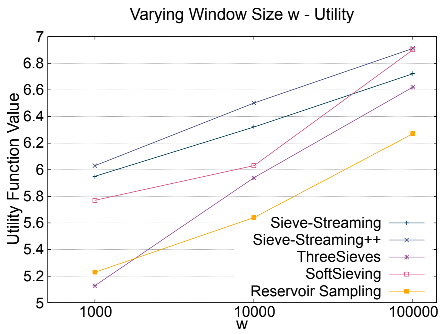
The window size is expected to influence mainly the utility value. Different window sizes show how quickly the approaches build a high-quality summary and how much they improve on their summaries over time. The total latency stays consistent for most approaches when varying w, with the exception of ThreeSieves for $w = 10000$ (Figure 5.6a). The local processing times (Figure 5.6b) of Sieve-Streaming, Sieve-Streaming++, and SoftSieving show a decrease when increasing w. The cause for this can be the increase in the number of sieves that have completed their summary and no longer require processing over the



(a) Total processing latency



(b) Local processing time



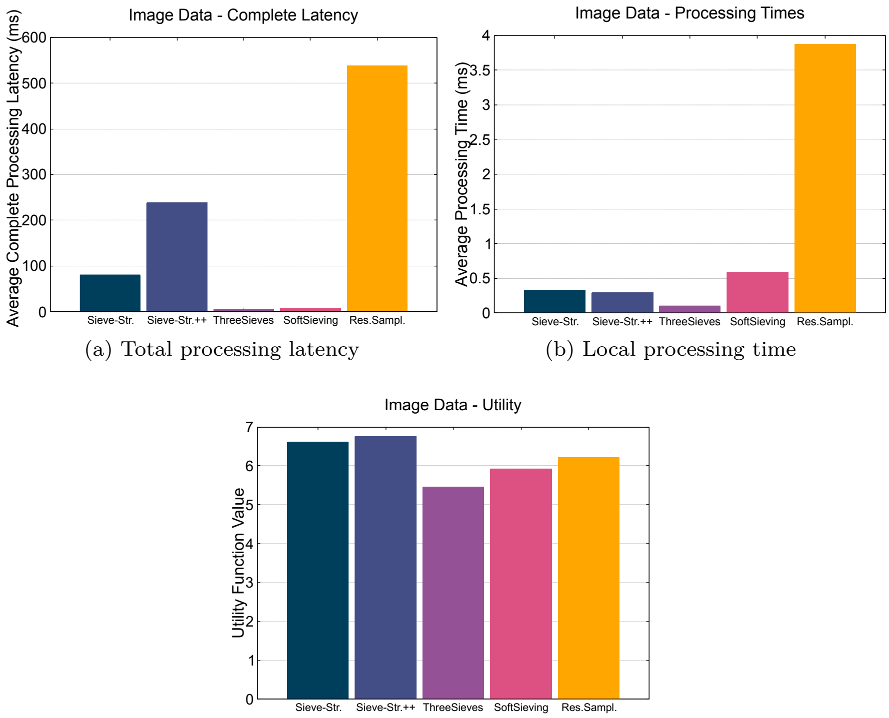
(c) Utility function value

Figure 5.6: Varying window size ($w = 1000$, $w = 10000$, $w = 100000$).

window. For the utility (Figure 5.6c), there is an increase for larger w for all approaches. Interestingly, this includes reservoir sampling. This may indicate that for our dataset, the data changes over time, and evenly distributed sampling naturally includes these changes. SoftSieving achieves higher utility as w increases such that it reaches the utility of Sieve-Streaming and Sieve-Streaming++ for $w = 100000$ while constantly outperforming ThreeSieves. Overall, we see stagnant latency and local processing times and increasing utility with increasing w for all approaches. Both SoftSieving and ThreeSieves get closer to the utility values of SieveStraming and SieveStreaming++ for $w = 100000$, while SoftSieving also works well for smaller w.

Image Dataset

As next, we discuss the results for the images dataset using the standard parameters and the active set utility function. For the complete processing latency (Figure 5.7a), the approaches show a similar distribution to the sensor data, with the exception of Reservoir Sampling. When looking at the local processing times (Figure 5.7b) it is apparent that the cause for this is the slow sampling in the respective bolt. The utility values (Figure 5.7c) show SoftSieving being lower than Reservoir Sampling, with ThreeSieves being the lowest by a large margin. This shows the impact of tuning T, or parameters like ϵ. The standard parameters work well for the telemetry dataset, but here, both approaches



(a) Total processing latency (b) Local processing time

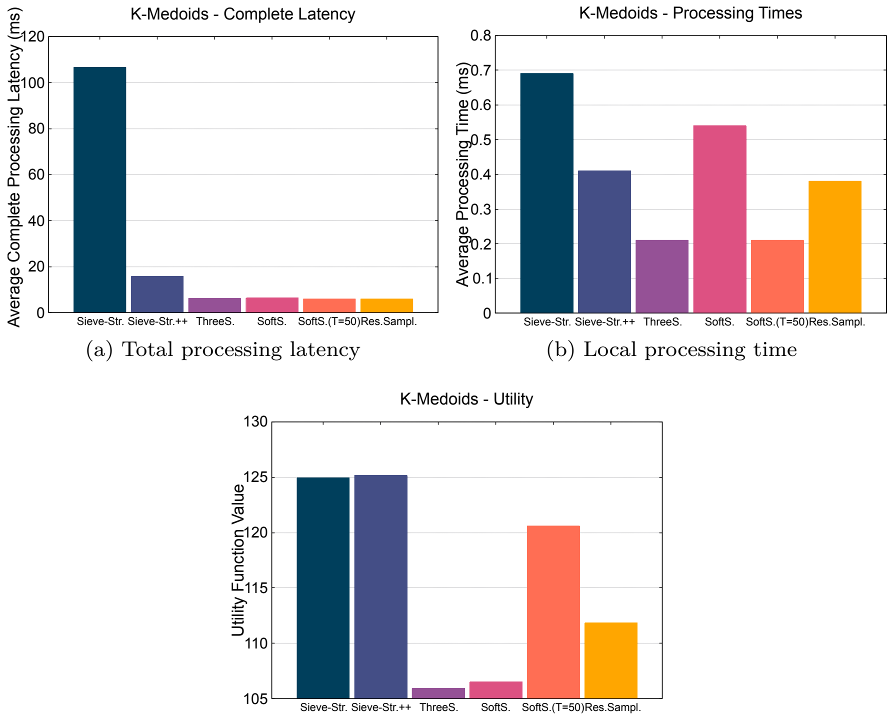
(c) Utility function value

Figure 5.7: Image dataset (standard parameters).

struggle to set appropriate thresholds. We saw the worst case of a utility of 0 for ThreeSieves earlier, and similarly, both approaches are surpassed by random sampling in utility since they are too restrictive with their thresholds. We conclude that the choice of the dataset can influence the latency and processing times as well as the utility.

K-Medoids Utility Function

We evaluated the k-medoids utility function on the telemetry dataset with the standard parameters. K-medoids requires samples, thus, all topologies include a SamplingBolt, collecting sample sets of size 50 to use for the per window utility calculations. The complete processing latency (Figure 5.8a) is low for all approaches except Sieve-Streaming. This is because the effects of having more sieves, and therefore more utility function evaluations, are stronger with k-medoids. The local processing times (Figure 5.8b) of SoftSieving are higher than ThreeSieves. Since the total latency remains low, this indicates effective partitioning of the data stream. SoftSieving with $T = 50$ was included to highlight the importance of T. It achieves processing times similar to ThreeSieves. The utility of SoftSieving (Figure 5.8c), although higher than ThreeSieves, is low compared to the other approaches. However, the high utility for $T = 50$ shows that this is a result of the choice of T. In conclusion, the results are comparable to active sets, but the choice of T can greatly influence our approach.



(a) Total processing latency

(b) Local processing time

(c) Utility function value

Figure 5.8: K-medoids utility function (standard parameters).

5.7 Summary

In this chapter, we investigated the problem of continuously extracting core-item–based summaries from a data stream. Specifically, we looked at an edge setting, where finding such summaries can save network load and speed up centralized applications that depend on the edge data. We proposed a new algorithm for data stream summarization using core items, called SoftSieving. It uses a fixed, low number of dynamic sieves and enables parallelized processing. The summaries are soft, meaning that core items can be swapped for ones with greater utility gain. We compared the performance to the state-of-the-art sieve-based algorithms in an extensive experimental evaluation and showed that our approach achieves acceptable balance between fast processing and high utility.

Chapter 6

Data Summarization for Approximate Query Processing

6.1 Introduction

In the previous chapter, we considered the computation of data summaries when working in resource-constrained streaming environments. However, data summarization can also be instrumental when working with vast amounts of data in a static setting. In this chapter, we present an approach for data summarization for the task of approximate query processing. The content of this chapter is based on our publication at the AIDM workshop was co-located with SIGMOD 2023 [62].

For data-intensive tasks under latency or resource constraints, approximate query processing can offer a remedy if exact answers are not imperative. This particularly applies to areas such as data exploration and visualization where the approximate answers can be used for discovering trends [169] or identifying parts of the data that need further processing through interactive analysis [54, 130]. To this end, in this chapter, we propose a versatile approach to lightweight, approximate query processing by learning compact but tunably precise representations of larger quantities of original tuples. Our approach lifts traditional processing of relational tuples to processing so-called **tuple bubbles**, or briefly put bubbles. Bubbles represent groups of tuples from underlying tables in a compact manner. In other words, bubbles are summaries of chunks of tuples, typically from one partitioned relation, but possibly from pre-computed results for frequent subqueries or primary-foreign-key joins. The bubbles are tunable regarding the compactness of enclosed tuples as well as the granularity of statistics and the way they are instantiated.

When a query arrives, an approximate answer is derived based solely on bubbles. We expect that there will be orders of magnitude fewer bubbles than tuples. Consequently, query processing can be vastly cheaper, enabling the processing of extensive datasets even for interactive tasks. In distributed environments, especially in so-called disaggregated settings where data shipping is mandatory, bubbles can deliver approximate query results in a bandwidth-saving manner and enable reliable dataset discovery in data lakes.

While there are multiple ways to succinctly summarize tuples in a bubble, like sketches [15, 50], sampling, or histograms [78, 101, 123], we propose the usage of **Bayesian networks** [68, 69, 155, 165] and **deep autoregressive models** [39, 166, 167], respectively. Even though Bayesian networks have been successfully applied for the task of answering count queries, i.e., cardinality estimation [68, 69, 155, 165], to the best of our knowledge, this is the first approach that employs them for AQP. Through the use of Bayesian networks, we produce estimates in a timely manner while efficiently representing dependencies between correlated attributes. Additionally, they allow the connection of estimates from different tuple bubbles, which is crucial for answering aggregation queries that involve joins. The usage of autoregressive models for cardinality estimation has been thoroughly investigated and shown to provide accurate query estimates within an acceptable time frame [39, 71, 166, 167]. The unsupervised model can represent complex data distributions and can answer queries involving any subset of column values. To apply it on heterogeneous datasets, we employ a per column lossless compression [38, 39]. To answer join queries over arbitrary tables of the dataset, the autoregressive model requires the complete join over the database tables for training.

As a starting point, we focus on how to represent bubbles and how queries are executed over these bubbles. However, ultimately, the goal is to provide operators semantically similar to standard operators in relational algebra, now at the level of bubbles. Doing so would leave traditional query processing paradigms intact, thus applicable. For the creation of bubbles, we focus for now on a simple horizontal partitioning schema, but as we will see in the experiments, even such partitioning already enables feasible approximate answers.

6.2 Contributions and Outline

In this chapter, we make the following contributions.

- We propose a solution for approximate query processing that performs the processing over groups of tuples, i.e., tuple bubbles, instead of individual tuples.
- We represent the bubbles through statistics using Bayesian networks and deep autoregressive models and create the final estimate considering queries with arbitrary number of joins and predicates.
- We thoroughly analyze the encountered challenges, and report the promising results.

The remainder of this chapter is organized as follows. In Section 6.3, we address the creation of bubbles as groups of tuples. Section 6.3.1 details the summarization of bubbles through Bayesian networks and deep autoregressive models. The querying and estimation using bubble summaries are discussed in Section 6.4. Section 6.5 gives an overview of the encountered challenges, while eventually, Section 6.7 summarizes the work in this chapter.

6.3 Creation of Bubbles

Bubbles can be derived from tables in various ways, ranging from basic horizontal partitioning, e.g., by primary key, to more involved approaches, such as identifying dependencies between tables and grouping similar tuples. The way bubbles are created influences the subsequent model creation. Therefore, if tuples are grouped based on the values of particular attributes instead of horizontal partitioning by primary key, only a subset of bubbles needs to be touched depending on the query predicates and the select clause. Consequently,

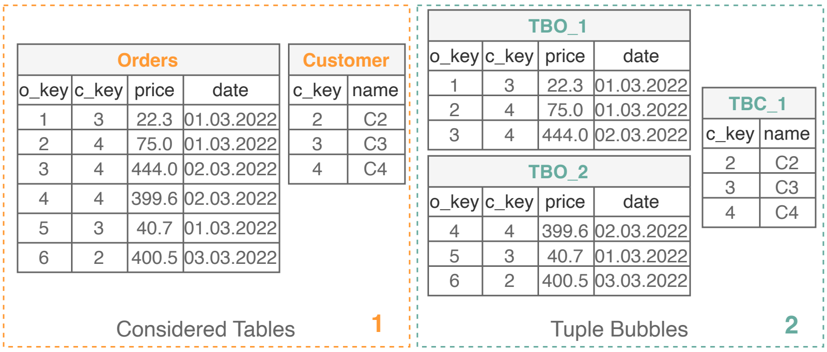


Figure 6.1: Example dataset (1) and bubbles for the tables of the dataset (2).

the choice of the grouping algorithm directly affects the system's capabilities in terms of estimation accuracy and execution time.

In this work, we create bubbles by horizontal partitioning of the underlying tables. To avoid partitioning tables that do not have many records, the user needs to set the partitioning parameter γ indicating the minimal number of records that a relation should have and the parameter k_{max} indicating the maximal number of bubbles that a table can be split into. An example of the bubbles creation is depicted in Figure 6.1. For the considered tables, *Orders* and *Customer* (Part 1 in Figure 6.1), and the parameter $\gamma = 3$, we will form three tuple bubbles (two for the table *Orders* and one for the table *Customer*) as shown in Part 2 of Figure 6.1.

For normalized databases, we expect many queries involving **foreign-key joins**. We can make use of this observation and create bubbles for joined partitions based on the foreign-key relations. For the *Orders* and *Customer* tables, the partitions for *Orders* (TBO_1 and TBO_2) will be joined with the partition for *Customer* (TBC_1) adding the *name* to the orders information. Although the upfront cost for the bubbles creation will be increased, by doing this, we immediately cover typical joins between two relations and, thus, can estimate query results more accurately.

6.3.1 Summaries of Bubbles

Once the records are logically assigned to bubbles, we create representative but compact models of the tuples contained in a bubble. One simple approach is to use histograms. Although they will be easy and cheap to create, they are effective only for queries per attribute and would not capture the dependencies between attributes. Furthermore, even more involved solutions that represent the data in chunks [101] cannot answer queries that involve joins, which drastically affects their practical usability. Thus, as a first comprehensive solution, we decided to utilize **Bayesian networks** and **deep autoregressive models** which can capture relationships between attributes. We leave the investigation of alternate solutions or hybrid variants over multiple different statistics for future work.

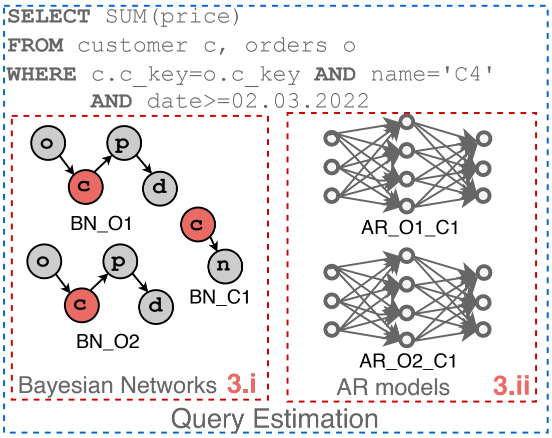


Figure 6.2: Bayesian networks (3.i) and autoregressive models (3.ii) for the tuple bubbles of Figure 6.1, and example query.

Bayesian networks

We instantiate one Bayesian network for each bubble to represent the conditional dependencies between the enclosed tuples. This means that for the three example bubbles in Figure 6.1 there will be three Bayesian networks (Part 3.i in Figure 6.2).

To identify a Bayesian network that closely matches the probability distribution between the attributes, we make use of the Chow-Liu tree structure learning algorithm [33]. To calculate the probability distribution for the nodes, we compute the conditional probability distribution for an attribute given the single parent attribute. The number of values for attribute A_i will be $card(A_i)^{p+1}$ where $card(A_i)$ is the cardinality and p is the number of parents which is one for our Bayesian network structure, i.e., $card(A_i)^2$ values need to be stored per distribution. Still, for attributes that have high value cardinalities a lot of values need to be stored. Thus, following related work for cardinality estimation [68,69,165], we store the exact probabilities for the k most appearing values of the attribute. For the remaining values, we employ binning by grouping them into b buckets where b can be varied according to the memory requirements. The less appearing values are discretized into equal-sized buckets. Every bucket b_i is identified by an integer id and together with the minimal and maximal value, the number of unique values for that range is stored. The algorithm for producing estimates from the Bayesian network is discussed in Section 6.4.

Deep autoregressive models

Deep autoregressive models are essentially a learned synopsis representing the relational table in an unsupervised manner. They efficiently estimate the joint distribution where given a predefined attribute ordering for the considered table, the model output contains the density for each attribute value conditioned on the values of the prior attributes.

To answer join queries over arbitrary tables from the dataset, the autoregressive models require the complete join result for training. For the example in Figure 6.1, there will be two models created (Figure 6.2 Part 3.ii), one for the join between the bubbles TBO_1 and TBC_1 and the other for the join between TBO_2 and TBC_1.

The size of the autoregressive model is directly impacted by the number of unique values of the attributes in the training dataset. More specifically, when using one-hot encoding, the logits from the input (output) of the model correspond to the conditional probabilities of the attribute values. Similarly, when using embedding, the size of the embedding matrix depends on the dimension defined by the distinct values. To limit the impact of highly heterogeneous attributes and enable the creation of models, we use a per-attribute lossless compression [38, 39]. The compression splits the values from a single attribute into multiple values for n subattributes. The size of the embedding of non-compressed attribute values is always much larger than the sum of the embedding sizes of the individual compressed attributes. The compression is lossless as a value is mapped to a unique combination of subvalues—and it can be reconstructed. With compression, the model is no longer impacted by the heterogeneity of the columns. The estimation is now done over the subattributes instead of the original attributes. Although this impacts the accuracy and execution time of the model, without it, the models cannot be applied to even datasets of modest size.

In addition to the requirement of training over the actual join, the values from non-numeric columns should be mapped to integer values, preferably in consecutive order. The mapping is necessary as the deep autoregressive model and the per-attribute compression can only be performed over numeric data. The need for these mappings motivated us to investigate improvements over the autoregressive model, resulting in the creation of Grid-AR introduced in Chapter 7.

6.3.2 Query Processing

When a user issues a query, we access the appropriate bubbles and, with that, the summaries, i.e., Bayesian networks or autoregressive model, relevant to the posted query. We refer to the derived queries over bubbles as *substitute queries*, as they are executed instead of the original query. Since we employ horizontal partitioning over the table keys for creating the bubbles, the attribute ranges will not be unique between different bubbles. Thus, to estimate a query we will need to access all bubbles of the involved tables. For now, we propose that the query answer is estimated by creating as many substitute queries as there are combinations of the bubbles for the relations of the query. For clarification, let us consider the example query from Figure 6.2. There are three bubbles, two representing the *Orders* table and one for the *Customer* table. The query performs a join between these relations and to answer it with our created statistics we would need two different queries, one joining TBO_1 with TBC_1 and the other TBO_2 with TBC_1. The estimate is then computed following the procedure in Section 6.4.3.

Evidently, if there are many tuple bubbles that should be considered for a given query, there will be many substitute queries that will need to be executed. In particular, if bubbles are created per relation, for a query that has l relations that can be represented by $k_{max_1}, \ldots, k_{max_l}$ tuple bubbles, where k_{max_i} is the number of bubbles for relation R_i, then $\prod_{i=1}^{l} k_{max_i}$ substitute queries would need to be estimated. Although the bubbles are represented through statistics and the computation will be much more efficient than considering all records from the relation, the required transformations and preprocessing for the substitute queries will increase the query processing time. To mitigate these effects, instead of selecting all bubbles for a query we can pick σ bubbles where $1 \leq \sigma \leq min(k_{max_i})$, balancing between the estimation accuracy and the execution time. For a small value of σ, to make sure that the chosen bubbles hold tuples that satisfy the query predicates, we can store an *additional compact index* for the bubble attributes.

To combine the substitute query estimates and obtain the final query estimate $est(q)$ we employ the following formula for `SUM`, `COUNT`, and `AVG` queries.

$$est(q) = \sum_{i=1}^{m} est(qs_i) * weight_i \tag{6.1}$$

The parameter m is the number of substitute queries, $est(qs_i)$ is the estimate for the query qs_i. The parameter $weight_i$ will be 1 for `SUM` and `COUNT` queries. For `AVG` queries $weight_i = N_{qs_i}/N$ where N_{qs_i} is the number of results of the query qs_i and N is the total number of results from all relevant queries. A substitute query is relevant if it has at least one result. For `MIN` and `MAX` queries, the final estimate will be the minimal or maximal estimate from the substitute queries, respectively.

6.4 Estimation over Bubbles

As we have seen, processing an aggregation query requires selecting appropriate bubbles, aligning them in substitute queries, and aggregating the obtained results. Below, we discuss the estimation algorithms over per-bubble summaries, for queries involving selection predicates. Then, we analyze how the summaries can be utilized for answering aggregate queries. Finally, when using Bayesian networks, we show how different bubble summaries can be connected for answering join queries.

6.4.1 Estimation over Single Bubble Summaries

Bayesian networks

Assume that we are working with the summary of one of the bubbles representing the attributes of the relation *Orders* as shown in Part 3.i of Figure 6.2 and that we want to perform inference over it. When the summaries are represented by Bayesian networks, the process of estimating the probability of some subset of the attributes A given some assignment of other attributes E (evidence), i.e.,

$P(A|E)$, is called *probability inference*. To compute this, we have to marginalize the joint probability distribution for all attributes that do not appear in A and E. We make use of two inference algorithms. For exact inference, we use the *variable elimination* algorithm [68, 69, 165]. For faster but approximate estimation, we utilize *progressive sampling* [165, 167].

Variable elimination is an efficient exact inference algorithm that can be applied to any kind of network [171]. The algorithm performs marginalization over the joint probability efficiently by moving the sum and product operations. It avoids repetitive computations and it operates on much smaller factors. For the example bubble representing the table *Orders* (Figure 6.2), to infer over the attribute *price* (p), the algorithm can be applied as:

$$P(p) = \sum_{o} P(o) \sum_{c} P(c|o) * P(p|c) \sum_{d} P(d|p) \tag{6.2}$$

By moving the summations inside, the algorithm reduces the computation by operating over much smaller relations. The algorithm can be further improved by eliminating the nodes from the network that are not required for obtaining a marginal distribution [68, 69].

The main idea of **progressive sampling** is to draw samples in iterations, progressively, such that the attributes will be traversed one at a time following a fixed ordering [165, 167]. In other words, the samples from the first attribute allow us to focus in a more relevant part of the second attribute and so on. To see how this works, consider a single Bayesian network that models the table *Orders* as P_O. Let us assume that the user issues a `COUNT` query q with arbitrary predicates $R(A_1), ..., R(A_4)$ that can represent any regions for the attributes of the table *Orders*. The probability for the given query can then be expressed as:

$$P_O(q) = \prod_{i=1}^{4} P_O(A_i \in R(A_i) | Par(A_i) \in R(Par(A_i))) \tag{6.3}$$

where $Par(A_i)$ is the parent node of the attribute node A_i and $R(Par(A_i))$ is the query region, defined by the query predicate, for that attribute. Consequently, to estimate the query probability, we need to estimate the conditional probability of the product for every attribute in Equation 6.3. To efficiently compute this, the algorithm utilizes Monte Carlo approximation based on a sample S of $R(Par(A_i))$ as $\frac{1}{S} \sum_{s \in S} P_O(R(A_i)|s)$. To execute the algorithm, first, the attributes are ordered as $A_1, ..., A_4$ where A_1 is the root of the Bayesian network. For the first attribute A_1, we can immediately obtain the probability $P_O(R(A_1))$ and use it to generate a sample S_1. For every next attribute, the samples for the parent have already been generated since the attributes follow a fixed ordering. Using the samples we can obtain a distribution that approximates $P_O(A_i|R(Par(A_i)))$. This distribution is used for the estimation and also for generating samples for the following attribute. This procedure is repeated for the remaining attributes. Using the estimates, $P_O(q)$ can be obtained by taking their product.

Deep autoregressive models

The autoregressive model can be used to calculate the probability for a given bubble consisting of d attributes $A = [A_1, A_2, ..., A_d]$, such that it estimates:

$$P(A) = P(A_1)P(A_2|A_1)...P(A_d|A_1...A_{d-1}) = \prod_{d=1}^{D} P(A_d|A_{<d}) \tag{6.4}$$

Answering equality predicates is straightforward, as the logits for the defined predicates are directly retrieved from the model. To deal with range queries, the autoregressive model employs the same technique introduced and explained for Bayesian networks, i.e., *progressive sampling* [167]. The induced bias of the progressive sampling is corrected with importance weighting. As in previous work, to answer queries that do not involve all the attributes in a learned relation, we apply wildcard skipping [167]. During training, a random subset of attributes will have their values replaced with special tokens denoting wildcards. During prediction, the input will consist of the values for all the predicates in the query plus the special tokens for the wildcard predicates.

6.4.2 Estimating Aggregate Queries

For both approaches, cardinality (`COUNT`) queries can be directly computed by multiplying the obtained probability by the size of the considered bubble. For the other aggregation functions, we need to obtain probabilities for the respective aggregation attribute and perform the aggregation function over the attribute values.

Variable elimination

As an output of the Bayesian network, when utilizing the **variable elimination** algorithm, we obtain the probability of an assignment of one of the attributes (A_i), conditioned on one or multiple attributes (referred to as evidence E), i.e., $P(A_i|E)$. To create an estimate, the probabilities for the respective aggregation query need to be transfered. Since in the Bayesian networks we employ binning for some attributes, we need to adjust the computations to consider ranges instead of single values. A range is identified by a minimal and a maximal value together with the number of distinct elements for the range. Consequently, we would consider the minimum or maximum when working with `MIN` or `MAX` queries, respectively. For `SUM` and `AVG` queries, the average value for the range is computed and used in the computation for the query. Although this preliminary approach for ranges provides appropriate results, it has potential for improvement.

Progressive sampling

When using the **progressive sampling** algorithm, for both approaches, to apply the algorithm we first need to impose an ordering over the attributes of the respective bubble. Since our approach will work with any aggregation query with an arbitrary number of predicates, it may happen that some of the attributes used in the predicates are after the aggregation attribute in the respective attribute order. Consequently, the probabilities for the values of the aggregation attribute will not be affected by the samples of the predicates.

To overcome this, when any aggregate query except `COUNT` arrives, we will generate r additional queries where we sample values from the aggregation attribute and use them as predicates in addition to the existing query predicates. Then, the obtained probability for every such query will be multiplied by the bubble size to obtain the cardinality of the respective query. If the cardinality is greater than zero, the sample aggregation value for the respective query will be used in producing the final estimate. For `MAX` and `MIN` queries, we will take the maximum and minimum, respectively, from all the qualifying aggregation attribute values. For `SUM` queries, we will take the sum of the attribute values. For `AVG` queries, we will make use of both the estimated cardinality of the additional query and the sample value to produce the final estimate. It is important to keep the value for r small, usually less than ten since otherwise, the estimation time for the algorithm will be drastically increased.

For the autoregressive models, we are operating over the complete set of values for every attribute. Since r needs to be kept small, it is extremely unlikely to sample the exact aggregation attribute values that satisfy the query predicates. For that reason, instead of sampling r values for the aggregation attribute, we will sample r non-overlapping ranges. Furthermore, since we know r in advance, for every attribute in the bubble, we can bin the values into r non-overlapping ranges and keep track of the upper and lower bound of these ranges. This will enable us to always capture all the values for the aggregation attributes and produce more accurate results. By binning beforehand, we can create ranges that capture the values better. As estimates, we will again consider only the ranges for the additional queries that have cardinality greater than zero. However, as values, we will take the average value of the range. Naturally, the binning strategy will directly impact the estimates.

6.4.3 Estimating Join Queries

When the summaries are represented through **autoregressive models**, the model will be trained over the complete join of all joinable bubbles and there will be as many models as the number of different joinable bubbles. As a reminder, for our example bubbles in Figure 6.1, there will be two models, one for the join between TBO_1 and TBC_1 and the other for the join between TBO_2 with TBC_1 (Part 3.ii of Figure 6.2). Consequently, when a query such as the one in Figure 6.2 arrives, the estimation over the model will be performed according to the explanation in Section 6.4.2 and the final estimate will be generated using Equation 6.1.

However, when the summaries are represented as **Bayesian networks**, we do not have to compute the actual join. Instead, we can connect the results from individual networks to produce the final estimate. Let us consider that every relation in the database is represented by a separate bubble. This means that there are two bubbles summaries, one for *Orders* and another for *Customer*. Let us assume that the user issues the same query as in Figure 6.2 but with `COUNT` instead of `SUM`. To answer this query, we have to resort to the join uniformity independence assumption. Consequently, the estimate will be $3/6 * 1/3 = 1/6$

Algorithm 6 AQP over Tuple Bubbles

function ESTIMATERESULT(Q, TB, I_{TB}, σ)
  $TB_{rel} = \{\}$; $TB_Q = []$; $Q_s = []$; $Q_{est} = []$
  $TB_{rel} = matchingBubbles(Q.relations, TB)$ ▷ get all bubbles
  $TB_Q = extractTB(Q,\ TB_{rel},\ \sigma,\ I_{TB})$ ▷ select σ bubbles
  for tb_q **in** TB_Q **do** ▷ use bubbles instead of relations
    $Q_s.add(replaceRelations(Q.relations,\ tb_q))$
  end for
  for i **in** $|Q_s|$ **do** ▷ estimate query through summaries
    $S_{q_s} = TB_Q[i]$
    if $S_{q_s}[0].type =$ $'BN'$ **then**
      $order(S_{q_s})$
    end if
    $Q_{est}.add(estimate(Q_s[i],\ S_{q_s}))$
  end for
  return Equation 6.1(Q_{est}, TB_Q, $Q.type$)
end function

instead of 2/6 which means that the approach underestimates the result by 50%. Furthermore, if the query was asking for `SUM` it would be impossible to incorporate the customer name in the Bayesian network for *Orders*, clearly leading to wrong results.

To alleviate this problem, and produce more accurate estimates for queries involving an arbitrary number of joins and predicates, we propose to make use of the per bubble results by *incorporating the results from one bubble summary into the others*. To accomplish this, when building the bubble summaries, we include all attributes for the relation, together with the primary and foreign keys. Although the primary keys will not help us capture important correlations, they are necessary to enable the estimation over join queries. Capturing the primary-foreign key relationships enables us to connect the bubble summaries by using the shared attributes between them. When having more than one bubble summary that needs to be used for answering the query, we need to establish an appropriate ordering of them, making sure that every summary can be connected through at least one attribute with the preceding and successive summary. To connect the Bayesian networks, starting from the first network, we extract the probabilities for the attribute that is shared with the successive Bayesian network. When doing this, we take into consideration the relevant predicates from the query that can be applied on the network. For the example query, since it includes the tables *Customer* and *Orders*, both bubble summaries are selected. Then, we order them to form a *chain following the primary-foreign key relation*, where the network that holds the aggregation attribute is last in the order. This is required because the probabilities of the aggregation attribute are needed to estimate the result.

For the example `COUNT` query, since *price* is the aggregation attribute, the bubble summary for *Customer* will be the first one and the one for *Orders* the second one in the order. The summaries can be connected through the attribute *c_key (c)*. For the first bubble summary we will extract the probabilities for *c* by applying the query predicate `name='C4'`. This would return the value 4. Then, for the bubble summary of *Orders*, we will consider the predicate `c_key=4` together with the query predicate `date>=02.03.2022`. Rows 3 and 4 of Figure 6.1 satisfy these predicates.

The complete approach for AQP over tuple bubbles is depicted in Algorithm 6. As input, it receives the aggregation query Q, the tuple bubbles TB represented through Bayesian networks or autoregressive models, the index for the attributes of the bubbles I_{TB} and the parameter σ restricting the number of bubbles. Initially, the method identifies all bubbles that can be used for replacing the relations of Q (Algorithm 6, Line 3). Subsequently, using the index I_{TB}, it selects σ qualifying bubbles aligning them to directly replace the relations of the query (Algorithm 6, Lines 4–7). For every substitute query, the summaries will be extracted. If the summaries are represented through Bayesian networks, they will be first ordered. The summaries will then be used for estimating the query (Algorithm 6, Lines 8–14). The final estimate is produced by plugging in the substitute query estimates in Equation 6.1.

6.5 Challenges and Outlook

When comparing the autoregressive (AR) model with the Bayesian networks, there are two clear downsides of the AR model. First, the AR model requires mapping of the columns into numeric values and, with that, incurs additional preprocessing time. This mapping can be done by storing a dictionary encoding. Naturally, this will impact the memory occupancy of the estimator. Differently, one can also use a hashing function to map the values into integers. However, the new hashed values will most likely not be consecutive, which impacts the accuracy of the model as well as the size of the embedding matrices. As a second downside, to estimate join queries in the AR model, the model has to learn over the joined relations (or bubbles), further increasing the preprocessing time. Differently, in the Bayesian network, one has to store the join partners as foreign keys in the tables, and the computation of the actual join or sampling over the join to answer such queries is avoided. For estimating join queries for both models, one has to prespecify the join partners. Therefore, any issued join query where the join is not over the prespecified join partners cannot be answered.

Despite the clear high-level idea of lifting the query processing from tuples to bubbles, there are several open research questions. First, the creation of the tuple bubbles needs to be thoroughly investigated. For large datasets, joining tables upfront to create bubbles for the result might not be feasible. Horizontal partitioning can be problematic, too, as it does not consider dependencies between the attributes of a table and requires the investigation of all possible bubbles for a table. If we instead realize a partitioning approach that can appropriately capture attribute dependencies, we will be able to potentially create more meaningful and comprehensive bubbles. Additionally, if we take into consideration past query workloads for the creation of bubbles, we can create self-contained bubbles that store all the relevant records for a particular group of queries and with that provide more accurate estimates.

Furthermore, as it will be shown in our experiments, combining results from separate bubbles to form the final query estimate is challenging. Specifically for `MIN` and `MAX` queries, for now, we can only estimate the result based on the minimal or maximal value in the considered bubbles and not over all the bubbles that satisfy the query. This limits our results since the approach heavily depends on the chosen bubbles. Additionally, combining results for the substitute queries following traditional techniques produces estimates that although acceptable, can be further improved. The reason for this is the same as for the `MAX`/`MIN` queries although the results can be slightly improved considering basic statistics such as the bubble sizes. As one direction, we can consider storing compact index structures and representative thresholds for the individual bubbles and use them for better estimation.

Although the Bayesian networks and autoregressive models are able to capture the correlation between the attributes and with that provide promising results, other summarization approaches can be useful for different use-cases. In particular, when handling single tables, a structure such as PASS [101] can accurately represent the data while occupying minimal space. Moreover, even between the bubbles or within a single bubble we can store different statistics that are better for answering different queries.

6.6 Experimental Evaluation

We implemented the proposed approach in Python and for the autoregressive model we used PyTorch. The experiments for the autoregressive model were carried out on an NVidia GeForce RTX 2080 Ti GPU, and for the Bayesian network and the competing approaches on an Intel Xeon E5-2603 v4 CPUs @1.7 GHz, 128GB RAM. For the experiments, we report on *accuracy*, *estimation time (i.e., latency)*, and *memory*.

6.6.1 Competitors

We considered and evaluated the following approaches:

1. Our tuple bubbles approaches, set up in different flavors:
    - The bubbles summarized as Bayesian networks (**TB_BN**) representing the complete relations from the database.
    - The bubbles as autoregressive models (**TB_AR**) representing the full outer join of the database relations.
    - The relations are horizontally partitioned into k_{max} partitions, i.e., bubbles, represented as Bayesian networks. For estimation we use $1 \leq i \leq k_{max}$ bubbles for a relation (**TB_BN_i**).
    - The bubbles are joins between relations that are horizontally partitioned into k_{max} partitions, represented as autoregressive models. For estimation we use $1 \leq i \leq k_{max}$ bubbles for a join (**TB_AR_i**).
    - Tables are joined based on the foreign-primary key and there is one Bayesian network (**TB_BN_J**) per join result.

- Relations are horizontally partitioned into k_{max} partitions and the partitions are joined based on the foreign-primary key relations. For each join result, one Bayesian network is created, and i bubbles from the same pair of relations are used for estimation (**TB_BN_J_i**).

2. PostgreSQL 14 serving as a baseline for producing exact answers.
3. VerdictDB (**VDB**) [128] integrated in PostgreSQL and accessed through Java. We used scrambles of ratio 10% and 50%.
4. Wander join (**WJ**) [99] integrated in PostgreSQL and accessed through Java. The approach can answer only `SUM` and `COUNT` queries, so, when evaluating, these are the only considered queries.
5. **KD-PASS** [101] using the authors' implementation in C++ with standard parameters as proposed by the authors.
6. The **AQP++** [130] implementation of Liang et al. [101] using the proposed parameters.

KD-PASS and AQP++ can only answer queries without joins, meaning that they are applicable only on single table datasets.

6.6.2 Datasets

For the evaluation, we consider the following datasets.

The international movies database benchmark (**IMDB**) [95] that consists of information about movies represented in 21 tables. The dataset occupies 3.6 GB when represented as CSV files. We do not evaluate the approach that uses the autoregressive model (TB_AR) because of the size of the full outer joins between the tables. One possibility to make the autoregressive model applicable is to sample over the joins and with that reduce the join sizes [166]. For the evaluation we use the job-light queries and create additional aggregation queries (150 in total) with 2 to 5 joins and 2 to 5 predicates.

The **TPC-H** benchmark that consists of 8 tables. We generated 1 GB of data and created 150 aggregation queries involving 2 to 5 joins and from 2 to 5 predicates.

Intel Wireless Dataset [17] which is a single table dataset consisting of data from 54 sensors. It contains of around 3 million rows and has 8 attributes where all attributes are continuous. For our evaluation, we considered all attributes and we created 100 aggregation queries with 2 to 5 predicates.

In our current version of the approach, we do not consider queries with group by or nested subqueries. As an aggregation predicate for the examined datasets, we consider any of the numeric, integer or date attributes. For the autoregressive model (TB_AR), all the trained models have three layers with 512 neurons and the training time for the considered datasets was at most one minute. The attributes that have more than 50000 values are compressed into two subattributes. The input and output are embedded using an embedding of 32.

Table 6.1: Performance of the algorithms on TPC-H. The values for the TB_BN approaches are for PS and VE, respectively and for the TB_AR approaches the values are for PS.

	Q-error (PS/VE)				**Avg. Time**	**Memory**
Approach	median	95th	max	avg	ms	MB
PostgreSQL	1.0	1.0	1.0	1.0	1124.9	1399
TB_BN	3.9 / 3.3	4620.0 / 4555.6	5.3*10^4 / 5.3*10^4	1064.0 / 1049.4	**12.01** / 58.7	**4.8**
TB_BN_1	2.3 / 2.29	4193.7 / 3810.5	5.3*10^4 / 3.8*10^4	1101.9 / 872.1	**10.3** / 45	**4.7**
TB_BN_J	1.2/ **1.018**	2500.1 / 3100.6	**9740.0 / 9740.0**	**220.1 / 241.5**	**16.4** /160.4	**10.8**
TB_BN_J_1	2.02 / 2.006	2439.9 / 6477.7	3.1*10^4 / 1.1*10^6	861.4 / 1.1*10^4	**16** / 150	**17.3**
TB_AR	1.56	961	**9805**	**408.29**	41.3	36.9
TB_AR_1	3.42	475.22	4.1*10^5	5096.37	40.1	36.6
TB_AR_2	1.58	402.8	4.1*10^5	4013.15	40.01	73.2
TB_AR_3	1.36	**232.05**	4.1*10^5	3390.7	40.01	109.8
VDB 10%	1.18	2*10^6	1.1*10^{10}	7.8*10^7	96.1	147.5
VDB 50%	1.02	1.7*10^6	1.1*10^{10}	7.7*10^7	874.1	359
WJ	1.1	4.9*10^6	9.7*10^8	3.2*10^7	143.3	702

Table 6.2: Performance of the algorithms on IMDB (only PS).

	Q-error (PS)				Avg. Time	Memory
Approach	median	95th	max	avg	ms	MB
PostgreSQL	1.0	1.0	1.0	1.0	$1.6*10^4$	7655
TB_BN_J	2.2	**106.0**	**1971.0**	**47.9**	**25**	**55**
TB_BN_J_1	6.2	2554.5	$4.9*10^4$	1870.3	**19**	89.3
TB_BN_J_3	4.7	1116.2	7109.8	317.6	**80**	108.8
VDB 10%	**1.18**	1971.0	$1.9*10^5$	3524.9	3255.6	395
VDB 50%	**1.02**	1224.0	$9*10^4$	1483.9	$1.4*10^4$	666
WJ	2.44	$1.7*10^8$	$1.7*10^9$	$6.5*10^7$	108	5216

6.6.3 Experimental Results

We use $k_{max} = 3$ and $\gamma = 500\,000$ for the horizontal partitioning of the tables, where k_{max} is the maximal number of partitions and γ the minimal number of records for a table to qualify for partitioning. For the binning in the Bayesian networks, we use between 60 to 200 buckets. For the categorical attributes, we directly store information about the 40 to 100 most common values and put the remaining values in buckets. For the binning in the progressive sampling we set r to 10. For the autoregressive model we evaluate the progressive sampling (**PS**) approach and for the Bayesian network we evaluate both, PS and variable elimination (**VE**). For PS, we always use 1000 samples. For the accuracy evaluation we used the q-error which is the relation between the true result and the estimate.

$$q\_error = max(\frac{true(q)}{est(q)}, \frac{est(q)}{true(q)})$$

Evaluating Join Queries

First, we report on results for TPC-H (Table 6.1) and IMDB (Table 6.2). For these datasets, we did not consider KD-PASS and AQP++ since they cannot answer queries involving joins. When analyzing the results for the TPC-H dataset, it is evident that the opponents have unacceptably high average and maximal errors. Additionally, when considering the execution time and memory, both flavors of our approach drastically outperform the competitors. Concerning the different versions of our approach, building bubbles per table represented as Bayesian networks (TB_BN) yields the worse results. However, if we partition the tables horizontally and consider only one partition per table for answering the queries (TB_BN_1) we achieve estimates with acceptable quality, estimation time and memory. To better capture the join queries involving more than two tables, we let the bubbles represent complete foreign-primary key joins and summarize them using Bayesian networks (TB_BN_J). This version produces the best estimates overall, where the VE algorithm has the best median accuracy. Considering Bayesian networks representing bubbles where the tables are first partitioned then the partitions are joined where one bubble per join is used for estimation (TB_BN_J_1), it is observable that the accuracy is significantly worse for the higher percentiles, indicating that the connection of the results can be further improved.

When the bubbles are represented as autoregressive models (termed as TB_AR and TB_AR_i), as depicted, the approaches produce good estimates, especially when considering the q-error for the higher percentiles. As the number of bubbles increases, the results improve. Thus, the bubbles approach is promising when we also take into account the reduced pre-processing time per bubbles than for the complete database.

Guided by the results for the TPCH dataset, for IMDB (Table 6.2), we consider only the PS algorithm and bubbles summarized as Bayesian networks that represent foreign-primary key joins (TB_BN_J) and partitions that are joined (TB_BN_J_i). The TB_BN_J approach again drastically outperforms the competitors in terms of accuracy, execution time, and memory. However, for the partitioned variant, using only one bubble per join (TB_BN_J_1), will not provide appropriate results in terms of accuracy. The main reason for this are the `MIN` and `MAX` queries since to answer them we always return the minimum or maximum for the investigated bubbles not considering the data from the other bubbles. However, using three bubbles per join already provides results of acceptable quality in satisfactory time while requiring drastically less disk space than the competitors. Although the optimal solution for creating the final query estimate is an open problem, the initial results support the conclusions that moving the processing from per tuple to per bubble can be highly beneficial.

Evaluating Single Table Queries

The results for the Intel dataset are shown in Table 6.3. Since the queries do not involve joins, we can consider KD-PASS and AQP++ in the evaluation. When tuple bubbles represent the 3 partitions of the table, we evaluate the approach when using 1, 2, or 3 bubbles for estimation, denoted as TB_BN_i and TB_AR_i, respectively. Although the approach where the bubbles summarized as Bayesian networks represent the complete table (TB_BN) has comparable median accuracy to the best competitor WJ, the estimates for the higher percentiles and the average are unsatisfactory. This is not unexpected since our approach uses buckets to represent continuous attributes. Naturally, this negatively affects the estimation. However, our approach requires drastically less disk space than most competitors. Additionally, the estimation using the PS algorithm, for both approaches, can produce results in the best time. When analyzing the results using bubbles summarized as Bayesian networks that represent partitions of the table (TB_BN_i), we can observe that the accuracy for the higher percentiles becomes worse. As the number of bubbles used for estimation increases, so does the median accuracy which for using all the created bubbles (TB_BN_3) even exceeds the accuracy of the TB approach. This same conclusion holds for the approach when the bubbles are summarized as autoregressive models, where when using all bubbles (TB_AR_3), we exceed the accuracy of the TB_AR approach. Although the required disk space and execution time increase, they are still better than most of the competitors.

Based on these results, moving the processing from tuples to bubbles is beneficial. However, connecting the results is still an open issue since clearly, heedlessly connecting estimates is not the optimal solution. When working with single tables, PASS can be considered for representing the summaries.

Table 6.3: Performance of the algorithms on Intel. The values for the TB_BN approaches are for PS and VE, respectively and for the TB_AR approaches the values are for PS.

	Q-error (PS/VE)				**Avg. Time**	**Memory**
Approach	median	95th	max	avg	ms	MB
PostgreSQL	1.0	1.0	1.0	1.0	153.5	201
TB_BN	1.24 / 1.1	44.6 / 43.7	2720.0.0 / 2964.0	141.1 / 109.9	**10.7** / 136.1	0.9
TB_BN_1	1.57 / 1.31	1240.3 / 2789.8	$7.7*10^5$	$1*10^4$ / $2.9*10^4$	**9.5** / 114.5	0.89
TB_BN_2	1.32 / 1.17	1331.3 / 1230.8	$7.7*10^5$	$1*10^4$ / $1*10^4$	20 / 230.3	1.8
TB_BN_3	1.17 / 1.12	1800.1 / 1179.0	$7.7*10^5$	$1*10^4$ / 8681.7	26 / 578.6	2.7
TB_AR	1.43	609.6	$5.9*10^4$	1682.6	17.3	4.6
TB_AR_1	2.96	1365.4	$1.3*10^5$	2597.3	16.6	4.2
TB_AR_2	1.5	467.7	$6*10^4$	1822.7	16.6	8.5
TB_AR_3	1.18	289	$5.9*10^4$	1068.4	16.6	12.8
VDB 10%	**1.003**	79.0	$5.9*10^4$	1265.3	55.2	22
VDB 50%	**1.001**	129.0	$5.9*10^4$	1508.1	135.6	55
WJ	**1.005**	**1.9**	**25.8**	**2.03**	155.3	90
KD-PASS	1.04	690.8	$2.2*10^4$	318.3	47	**0.1**
AQP++	1.042	809.9	$2.2*10^4$	327.4	52	**0.1**

6.7 Summary

In this chapter, we addressed the problem of performing AQP by lifting traditional per-tuple query processing to processing model-based descriptions of groups of tuples, referred to as tuple bubbles. To accomplish a concise but accurate description, we proposed the usage of Bayesian networks and autoregressive models and investigated their suitability to compactly represent the tuple bubbles and their ability to answer queries involving an arbitrary number of equality joins and equality or range predicates. We further detailed on encountered challenges and validate our ideas through a preliminary evaluation. In the following chapter, we will focus on better utilization of the autoregressive model by combining it with an additional grid structure to enable efficient handling of queries involving range and equality predicates.

Chapter 7

Data Organization for Cardinality Estimation

7.1 Introduction

In this chapter, we will focus our efforts on improving the utilization of deep autoregressive models for the task of cardinality estimation. Accurate cardinality estimates are a key ingredient to guarantee optimal query execution and to tune applications to reach peak performance. Traditional methods based on histograms [40,66,67,117] or sampling [96,98,173] often face challenges in providing precise estimates, especially for complex queries or skewed data distributions. Thus, to overcome these challenges, learned approaches, both supervised and unsupervised, have been successfully applied for cardinality estimation. Supervised algorithms trained on historical data offer the capability to understand complex relationships and patterns [39, 45, 73, 88, 106, 107, 127, 162]. Although supervised approaches capture complex attribute correlations for a small memory budget, they typically fail to adapt to query distributions outside of the ones used for training. Differently, unsupervised models learn directly from the data without requiring query workload generation [39, 62, 71, 73, 76, 165–167]. Since we want to develop an algorithm that does not depend on a given query workload but instead can learn directly from the data, we pursue exploring deep autoregressive models. Furthermore, we have already established the benefits of utilizing such models in our previous work of approximate query processing.

When analyzing autoregressive models for cardinality estimation [39,62,71,166, 167], we see that the number of distinct values per column in the tables directly impacts the *accuracy*, the *size of the estimator*, and the *prediction speed*. The **accuracy of these models** depends on the complexity of the data. Columns that have many distinct values are more likely to possess complex dependencies, making them inherently more challenging to learn. Furthermore, autoregressive models with many parameters can require significant **training time** and substantial **memory consumption**, which limits their applicability in resource-constrained environments. An increase in the dimensionality directly impacts the size of the embedding matrices, i.e., the model grows with the number of distinct values per column. Recent efforts to address this problem employ per-column compression [39, 166], which drastically reduces the models' size while it increases the interconnection complexity and leads to a small decrease in accuracy. However, even with per-column compression, storing the *mapping of column values converted to integer values*, as required by the autoregressive model, is still crucial to reduce the memory consumption.

Another concern with autoregressive models is the poor performance of **queries with range predicates**. Existing studies [166, 167] leverage progressive sampling to efficiently sample for answering such queries. However, due to the

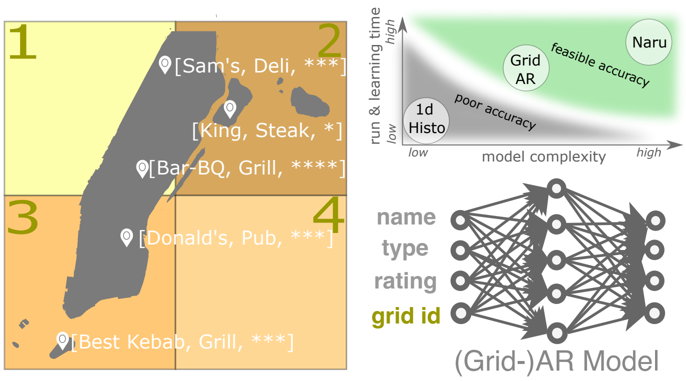


Figure 7.1: Left and bottom right: 2D four-cell grid over Manhattan's restaurant ratings and AR model with coordinates replaced by grid cells; Top right: space of possible solutions.

iterative nature of the estimation process, this approach leads to high execution times. Queries involving ranges appear frequently when dealing with temporal, spatial, and metric data. Such queries can be combined with equality predicates on categorical attributes, like in spatial keyword queries. Figure 7.1 illustrates such a scenario, highlighting points of interest like restaurants. Predicates on categorical attributes, such as restaurant names or types, typically check for equality, while attributes like restaurant location and opening hours are often used with range predicates.

To overcome the problems of autoregressive models, we propose Grid-AR, a hybrid structure that integrates a *grid structure* for columns typically used in range predicates and an *autoregressive model* for columns used in equality predicates. The grid acts as a preliminary filter for predicates, leading to the segmentation of data into simpler partitions that substitute the continuous attributes in the original data before the autoregressive model is trained.

Doing so promises several **benefits**. First and foremost, it allows fast prefiltering of tuples that need to be considered by pruning the search space upfront. Secondly, by substituting the ample space of possibilities with prefiltered partitions, the need for progressive sampling during estimation is completely circumvented, drastically speeding up the execution. Finally, such a hybrid structure avoids storing large embeddings and many dictionary mappings for the numerical columns, resulting in a smaller memory consumption. The suggested modification markedly improves the execution time of the model for both training and prediction, reduces memory consumption, and does so with minimal decline in accuracy. To validate the effectiveness of our cardinality estimator, we conduct and present a comprehensive evaluation considering state-of-the-art competitors using three benchmark datasets—demonstrating vast improvements in execution times and resource utilization.

Mixed data containing categorical and continuous attributes is often queried using **range joins**. An example is locating restaurants of type `deli` with better ratings than the restaurants of type `pub`. Such a query will result in a self join over restaurants with a join condition over the rating column. Range joins between two relations typically narrow down to three types of conditions: inequality, a point in interval, and interval overlap conditions. Related work [85, 136, 138] optimizes such queries for query execution rather than cardinality estimation. **We introduce a generalized definition of range join queries and propose an algorithm that employs Grid-AR to provide fast and accurate cardinality estimates for such queries.** Thus, with Grid-AR, we extended the possibilities of using learned models for a task that has only scarcely been addressed for relational databases so far.

7.1.1 Contributions and Outline

In this chapter, we make the following contributions.

- We propose a cardinality estimator, termed **Grid-AR**, by combining a grid and an autoregressive structure.
- We develop an efficient algorithm for fast execution of queries over single tables through employing grid structure prefiltering while reducing the memory overhead of storing additional per-column value mappings.
- We present an algorithm employing Grid-AR for estimating the cardinality of range join queries.
- We perform a comprehensive experimental study over one synthetic and two real-world datasets.

The remainder of this chapter is organized as follows. The need for creating a hybrid structure for cardinality estimation and the details of our proposed approach termed Grid-AR are presented in Section 7.2. In Section 7.3, the algorithm that utilizes Grid-AR for cardinality estimation over single table queries is explained. Section 7.4 details the usage of Grid-AR for estimating range-join queries. The results of a comprehensive experimental evaluation are presented in Section 7.5, and the chapter is summarized in Section 7.6.

7.2 Boosting Learned Cardinality Estimation

Grid-AR uses data partitioning to remedy a key problem of autoregressive models for cardinality estimation. Despite its name, Grid-AR is conceptually compatible with alternative methods to partition the data space. We use grids due to their fast processing speed and low memory consumption. While various tree-structured indices, such as kd-trees or quadtrees (cf., [53, 140]), also partition the space in disjoint partitions and may provide a better grouping of points, a uniformly created grid can return the cell that contains a query point in constant time. Using grids for selectivity estimation is also very common, as one- or also multi-dimensional histograms are basically an imposed grid partitioning on the data space [4, 21, 132]. However, the efficiency of a grid can be affected by data skew and varying object densities across the cells. Thus, we explore creating *distribution-aware grid cell boundaries* in addition to adhering to a uniform division.

While many alternative data-driven models, such as Bayesian networks exist, the results of [167] suggest that autoregressive models provide better accuracy.

Although highly accurate, autoregressive models come with two main flaws. Queries with range predicates over autoregressive models are based on an iterative process, i.e., progressive sampling. Sampling yields large execution times and is negatively influenced by the increase in the number of columns used in query predicates. For instance, upon analyzing the execution over the *Flight* dataset, we have observed that **doubling the number of query predicates** from 2 to 4 results in almost **doubling the average estimation time**, i.e., 440 to 970, and 390 to 860 milliseconds for the models without and with compression, respectively. Moreover, due to the nature of machine learning models, the input to the model must be an integer. Hence, we also necessitate an integer mapping of the values. Although appropriate for categorical data, employing a dictionary mapping leads to substantial memory consumption in instances involving continuous data with high skewness. For the *Flight* dataset, as expected, the total size of all dictionaries increases linearly with the number of columns used in the estimator. We aim to fine-tune the estimator's structure to tackle an inherent challenge associated with progressive sampling in queries with range predicates, all while maintaining a compact memory footprint.

To ease both the creation and the querying over ranges of the autoregressive model for the task of cardinality estimation, we propose a novel hybrid structure called **Grid-AR**. Consider a table T with columns $C = \{c_1, c_2, \ldots, c_n\}$. We distinguish between two disjoint sets of attributes:

- $CR = \{cr_1, cr_2, \ldots, cr_k\}$ where $CR \subseteq C$ are columns typically involved in range predicates.
- $CE = \{ce_1, ce_2, \ldots, ce_l\}$ where $CE \subseteq C$ are columns typically involved in equality predicates.

This way of organizing the columns is based on an observation that columns that store continuous data, e.g., temporal or spatial data, are likely to be associated with range predicates, i.e., filtering by a range of values. Conversely, text attributes are more likely to be included in equality predicates, implying exact matches. First, we build a grid based on the CR columns, and then, using the grid cells together with the CE columns, we train the autoregressive model.

7.2.1 The Grid Structure

To create the **|CR|-dimensional** grid structure of Grid-AR, we utilize the values of the columns in CR to superimpose a grid on the data. For each dimension (column), we split the data into m buckets. To adequately capture the diverse characteristics of individual columns, the number of buckets differs for each column. A grid cell is represented by the boundaries of the buckets of every dimension. We consider two ways of creating buckets, one by simply following the uniformity assumption and the other by considering a given per-column data distribution. In the following, we will use the terms columns and dimensions interchangeably to refer to the columns of the table that represent the dimensions of the grid structure.

Uniform grid: In the first option, we group the column values, assuming the uniformity assumption. More specifically, within each dimension, the grid is partitioned into evenly spaced buckets spanning from the minimum to the maximum value. For every dimension i, there will be m_{cr_i} buckets that the column values will be organized into. The size of each bucket is computed as

$$bucket\_size_{cr_i} = \frac{max(vals_{cr_i}) - min(vals_{cr_i}) + 1}{m_{cr_i}}$$

where $vals_{cr_i}$ are the distinct values of the column cr_i. Thus, a column cr_i with value val_{cr_i} will be mapped to a bucket based on the equation:

$$bucket_{cr_i} = \left\lfloor \frac{val_{cr_i} - min(vals_{cr_i})}{bucket\_size_{cr_i}} \right\rfloor$$

A tuple from the table is matched to a grid cell by considering the buckets to which the values of the tuple belong. This way of creating the grid structure requires **storing** only a few necessary information: the min and max value per dimension, the buckets' boundaries per dimension, the buckets' size, and for every grid cell, the min and max value per dimension for the qualifying tuples.

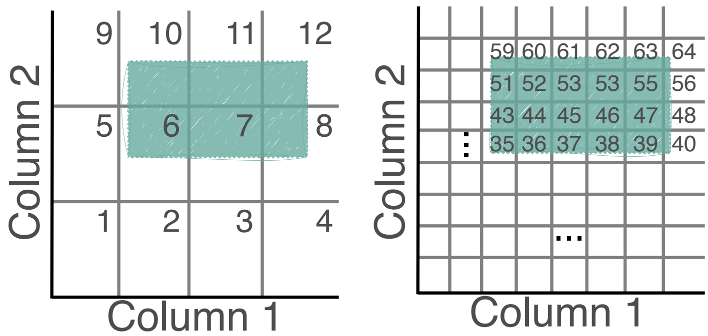
CDF-based grid: As an alternative to the grid creation above, we consider a grid tailored to the data distribution based on the cumulative distribution function (CDF) for every dimension. Instead of dealing with a direct value for each column val_{cr_i}, we consider the model output $f(val_{cr_i})$ representing the CDF of the value val_{cr_i}. In other words, the model responsible for computing the per dimension CDF will receive as input the value val_{cr_i} and will output the fraction of points with values that are less or equal to val_{cr_i}. In Grid-AR, we utilize a decision tree regressor [19] as a powerful tool for statistical modeling and learning the distribution of every column of the table. Unlike traditional decision trees for classification, regression trees are designed to predict continuous numerical outcomes. The construction of a regression tree involves recursive splitting of the dataset based on selected features and corresponding thresholds, ultimately leading to decision nodes and leaf nodes. At each decision node, the dataset is partitioned into subsets, guided by criteria such as mean squared error or variance reduction, with the goal of minimizing the variance of the target variable within each subset. The model accuracy improves by incrementing the number of levels; however, this increases the estimation time. The variance for a node n is calculated as

$$\mathrm{Var}(D_n) = \frac{1}{|D_n|} \sum_{i \in D_n} (y_i - \bar{y}_n)^2$$

where y_i is the target variable, $|D_n|$ is the data count, and $\bar{y}_n$ is the mean of y in node n. Predictions at leaf nodes ($\hat{y}_n$) are determined as

$$\hat{y}_n = \frac{1}{|D_n|} \sum_{i \in D_n} y_i$$

Model accuracy is assessed through the mean squared error over the actual and predicted value. These equations capture the essence of regression trees, highlighting their role in effective data partitioning, prediction, and model evaluation. For the grid structure of Grid-AR, we need to **store** the information



(a) Grid with less cells (b) Grid with more cells

Figure 7.2: The effects of querying over a grid index with less cells (left) and more cells (right).

that makes up the regression trees, the number of buckets per dimension, together with the minimal and maximal value for every dimension in all grid cells. For Grid-AR, a column cr_i with a value val_{cr_i} will be mapped to a bucket based on the equation:

$$bucket_{cr_i} = \lfloor f_{cr_i}(val_{cr_i}) \rfloor * m_{cr_i}$$

where m_{cr_i} is the number of buckets that the values of cr_i will be grouped into and f_{cr_i} approximates the CDF value of val_{cr_i}.

No matter the choice of the grid structure, a tuple t, considering only the columns in CR, corresponds to a grid cell id identified with:

$$gc_{id} = [bucket_{cr_1},\ \ldots,\ bucket_{cr_k}]$$

The organization of cells and the generation of grid cell ids is carried out through a depth-first traversal of cells along the dimensions.

CDF grids, in comparison to uniform grids, create splits according to the data distribution. The resulting split boundaries are not equally spaced across the minimum and maximum values of the dimension but are instead based on the cumulative distribution function (CDF). Consequently, regardless of the data distribution, a CDF grid ensures that each bucket contains roughly the same number of points, with more splits in dense regions and fewer in sparse regions. In contrast, uniform grids might be simpler to implement and computationally less complex, but their performance degrades with highly skewed or non-uniform data distributions, as highly populated buckets cause larger estimation errors. **While an equi-depth histogram may appear similar to a CDF-based split**, it can lead to the same value being allocated across several buckets. A CDF-based split guarantees that each data point is assigned to only one bucket. Given our goal of replacing multiple continuous values with a single cell id, we choose to utilize learned histograms.

Number of grid cells: In Grid-AR we can independently set the number of buckets m_{cr_i} for every dimension. Nevertheless, choosing a feasible number of buckets depends on the considered column and its values, while an optimal choice is clearly dependent on accurate assumptions on the query workload.

Determining the **number of buckets per dimension** means balancing a trade-off: When the grid has fewer buckets, the size of the estimator will be smaller, and the estimate can be produced faster since less cells are considered for answering the query. However, this comes at the cost of a lower accuracy as cell boundaries per dimension will be coarser, including tuples that potentially do not match the query predicate. In contrast, when the number of buckets is increased, the ranges are more tightly bound. Hence, for the same predicate more cells are considered, which increases the estimation time but captures the query predicates better. The difference between querying a grid with less and more cells is depicted in Figure 7.2.

In our experimental evaluation (Section 7.5.4), we show the effects of varying the number of grid cells. To achieve an optimal balance between the execution time, memory, and accuracy, for smaller datasets where a clear pattern in the grouping of the column values can be observed, we rely on the analysis of per-column values. However, when we cannot draw any conclusions about the number of buckets per column directly from the dataset, we apply a cost model for identifying the grid layout, similar to the one proposed by Nathan et al. [120]. We consider a sample query workload, and based on it, try to identify the optimal number of ranges per dimension to achieve the right balance between the time, memory, and accuracy.

7.2.2 The Autoregressive Model

In the **second step** of the creation of Grid-AR, we proceed with creating the autoregressive model. To perform the training, the autoregressive model, in addition to the CE columns, requires as input the grid cell ids. In other words, a new tuple is created by combining the grid cell id to which the tuple belongs together with the CE attributes of the tuple. The table is thus transformed from $cr_1, \ldots, cr_k, ce_1, \ldots, ce_l$, to $gc_{id}, ce_1, \ldots, ce_l$. With a designated column order, the autoregressive model is trained to comprehend the interdependent relationship between the values of a column and the values of the preceding columns in the specified order. Consequently, in this scenario, the model acquires an understanding of the dependencies between the grid cell ids and the remaining columns. As already outlined, to make the autoregressive model applicable to highly heterogeneous columns, we apply a per-column lossless compression [39]. A column qualifies for compression if it has more than a prespecified number of distinct values γ. The hybrid structure brings forth an important advantage. Using the grid structure, we avoid the storage of mappings of continuous columns to consecutive integers. This directly speeds up the model and drastically reduces the memory footprint.

7.2.3 Example for Grid-AR Creation

In Figure 7.3, we demonstrate the creation of our hybrid structure Grid-AR when considering information about restaurants depicted in the `Restaurant` table. We first distinguish between attributes that can be used in range predicates, i.e., continuous attributes, and attributes that can be used for equality

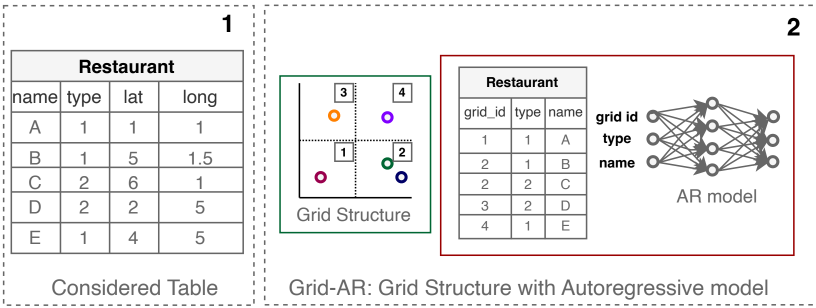


Figure 7.3: Creation of Grid-AR. Creation of the grid structure for the continuous attributes and an autoregressive model for the categorical ones together with the assigned grid cell id (part 2).

predicates, i.e., textual attributes. Thus, for constructing the grid, we establish a grid pattern over the location coordinates `lat` and `long`. For the sake of clarity, in this example, we partition each dimension into two regions, yielding a total of four grid cells. We assign a gc_{id} to every cell which is created from the per-dimension bucket ids. The grid cell id, together with the categorical attributes, `type` and `name`, are later used for training the autoregressive model. For instance, restaurant A has the coordinates $(1, 1)$ which align with the grid cell with id 1. As depicted in part 2 of Figure 7.3, the grid cell ids will be assigned to the respective tuples based on the columns used for creating the grid structure; in this example, the columns `lat` and `long`.

7.3 Estimating Single Table Queries

During querying, Grid-AR estimates the query selectivity using the estimates for the CR columns on the grid structure and the estimates for the CE columns on the autoregressive model. Algorithm 7 illustrates the querying process of Grid-AR for providing the cardinality for queries over single tables involving both range and equality predicates. When querying the structure, the algorithm divides the query into two subqueries: one involving columns used in the grid structure and the other for the autoregressive model, i.e., Q_{grid} and Q_{AR} (Algorithm 7, Line 2).

The first phase involves using the grid to identify cells overlapping with the query boundaries in Q_{grid} (Algorithm 7, Line 3). This is done by obtaining the grid cells that intersect with the query boundaries in each dimension (Algorithm 7, Lines 13–23). Let GC_i represent the set of cells that overlap with the query boundaries in the i-th dimension obtained with the method *gcs_per_dim*. The intersection is then applied to these sets to retain only the grid cells that qualify in all dimensions: $GC_{grid} = GC_1 \cap GC_2 \cap \ldots \cap GC_k$. In the end we obtain GC_{grid} representing the filtered set of grid cells that satisfy Q_{grid}.

Once the qualifying cells have been determined, the algorithm switches to the autoregressive model. In this step, the model receives the ids of the cells in GC_{grid} along with the remaining values for the columns in CE for which the

Algorithm 7 Grid-AR single table query

1: **function** ESTIMATE(*Query Q*, *Grid GRID*, *AR Model AR*)
2: $Q_{grid}, Q_{AR} \leftarrow split\_sq(Q)$ ▷ split Q according to columns
3: $GC_{grid} \leftarrow cells\_for\_query(Q_{grid}, GRID)$ ▷ query grid
4: $est_{AR} \leftarrow AR.estimate(GC_{grid}, Q_{AR})$ ▷ query AR-model
5: $card \leftarrow 0$
6: **for** $gc_i \in GC_{grid}$ **do** ▷ for each qualifying grid cell
7: $est_{AR_i} \leftarrow est_{AR}[gc_i.id]$
8: $est_{gc_i} \leftarrow cell\_estimate(gc_i, Q_{grid})$
9: $card \leftarrow card + est_{gc_i} * est_{AR_i}$
10: **end for**
11: **return** $card$
12: **end function**
13: **function** CELLS_FOR_QUERY(*Query Q*, *Grid GRID*)
14: $GC_{grid} \leftarrow \{\}$
15: **for** $dim \in Q.dims$ **do** ▷ for each dimension
16: **if** $dim == 0$ **then**
17: $GC_{grid} \leftarrow GRID.gcs\_per\_dim(Q[dim])$
18: **else**
19: $GC_{grid} \leftarrow GC_{grid} \cap GRID.gcs\_per\_dim(Q[dim])$
20: **end if**
21: **end for**
22: **return** GC_{grid}
23: **end function**
24: **function** CELL_ESTIMATE(*Grid cell gc*, *Query Q*)
25: $b_{min} \leftarrow max(gc.min, Q.min)$ ▷ min for each dimension
26: $b_{max} \leftarrow min(gc.max, Q.max)$ ▷ max for each dimension
27: $volume \leftarrow 1$
28: **for** $dim \in 0 \ldots (|b_{min}| - 1)$ **do** ▷ for each dimension
29: $volume \leftarrow volume * (b_{max}[dim] - b_{min}[dim])$
30: **end for**
31: **return** $volume/gc.volume$
32: **end function**

model is responsible (Algorithm 7, Line 4). More specifically, for each grid cell, the estimate is computed as:

$$est_{AR} = P(gc_{id}, ce_1, ..., ce_l) = P(gc_{id})...P(ce_l|ce_{i<l}, gc_{id})$$

Note that the estimation is a simple point density estimation in the autoregressive model, computed in a single forward pass in a batch for all the grid cells. This way of computation circumvents the need for multiple iterations when sampling with the progressive sampling estimation algorithm. As a result, we get an estimate for the cardinality of the query Q_{AR} for each grid cell.

The algorithm proceeds to estimate the query selectivity by considering the results from each cell (Algorithm 7, Lines 6–10). The calculation takes into account the cardinality of the autoregressive model and the grid cell selectivity. In the grid structure, the algorithm estimates the *overlap volume* between the query Q_{grid} and the grid cell boundaries for every dimension, detailed in the

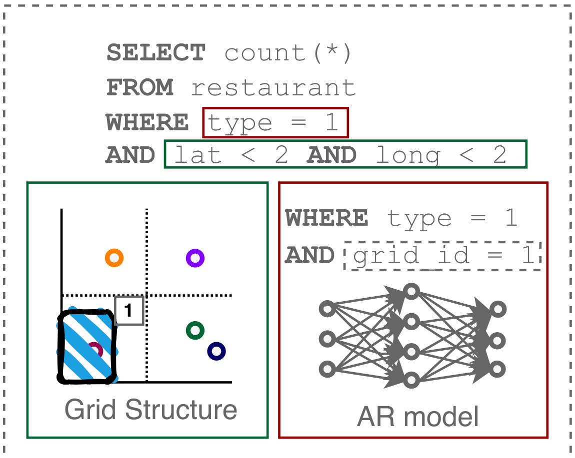


Figure 7.4: Querying of Grid-AR. For an example query, the Grid-AR parts responsible for the different types of query predicates are used for producing cardinality estimates.

function called `cell_estimate` (Algorithm 7, Lines 24–32). We return the fraction of overlap by dividing it with the volume of the whole grid cell as

$$est_{gc_i} = \frac{V(gc_i \cap Q_{grid})}{V(gc_i)}$$

As the query boundaries do not always completely contain a grid cell, **the overlap** indicates the proportion of grid cell points that should be included according to the query boundaries. Hence, if the query boundaries overlap with only 50% of the grid cell volume, the overlap value will be 0.5.

The volume obtained from the grid cell and the cardinality of the autoregressive model are multiplied and returned as the final estimate for the grid cell $est_{gc_i} * est_{AR}$ (Algorithm 7, Line 9). The individual estimates are summed up and returned as a result. The complexity of the single table estimate is $O(|GC_{grid}| * Cost_{AR})$ where $Cost_{AR}$ is a forward pass cost and depends on the number of layers of the autoregressive model.

7.3.1 Example for Querying Grid-AR

Figure 7.4 shows the execution of a simple query over the table `Restaurant` that filters the coordinates as well as the type of the restaurant. In the first step, we use the grid structure to filter out the possible grid cells that qualify for the range predicate. In this example, this will be the grid cell 1. The grid cell, together with the equality predicate *type* = 1, is forwarded to the model. The final estimate is calculated by using the overlap percentage of the query and the qualifying cells as well as the estimate $P(gc_{id} = 1, type = 1) = P(gc_{id} = 1) * P(type = 1|gc_{id} = 1)$.

Querying Grid-AR brings one significant benefit over a textbook autoregressive model. The computation of range predicates in the autoregressive model through progressive sampling requires n iterations of forward passes for each of the n columns in the query. With Grid-AR, the iterative computation is no longer necessary as the approach no longer uses progressive sampling since the grid structure already performs the range filter predicates. We will further demonstrate the effects on the execution time in our experiments.

7.4 Grid-AR Range Join Queries

The creation of the hybrid Grid-AR structure facilitates the execution of another type of queries commonly performed on data containing continuous attributes, namely **range join queries**. Existing research [85, 136, 138] concentrates on enhancing the execution of such joins and overlooks the aspect of estimating their cardinality.

A range join is an operation where two relations are joined based on an inequality, a point in interval, or interval overlap condition. An **inequality join** is the basic query form, where two attributes from two relations are compared to one another with any comparison operator except for equality. A **point in interval range join** contains a join condition characterized by predicates specifying that a value from a column in one relation falls within the interval defined by two values from a column in another relation. Let R and S be two relations. The point in interval join condition is expressed as $R.value \in [S.lower, S.upper]$, where $R.value$ is an attribute in R, and $S.lower$ and $S.upper$ define the lower and upper bounds of the interval and are values from an attribute in S. An **interval overlap range join** is a join operation in relational databases, defined by conditions expressed through predicates that require an overlap of intervals between two values, each derived from two relations. For two relations R and S this join can be expressed as: $R.lower \leq S.upper$ AND $R.upper \geq S.lower$ where $R.lower$ and $R.upper$ are the lower and upper bounds for an attribute in the first relation, and $S.lower$ and $S.upper$ are the corresponding bounds in the second relation.

We generalize these queries into a more generic form. In one query, we allow multiple join predicates of the form $f(R.c_i)\ \theta\ g(S.c_j)$, where $\theta \in \{\leq, <, \geq, >\}$ and f and g are arbitrary expressions over the attribute values. The expressions can range from a simple injective function $f(R.c_i) = R.c_i$ to a more elaborate one such as $f(R.c_i) = R.c_i * 2 + 100$.

7.4.1 Estimation Algorithm

Grid-AR is, by design, set up to perform cardinality estimation for range joins, as all the continuous attributes are part of the grid structure. We split the estimation in **two steps**.

Grid-AR query phase: First, we split the query into subqueries, isolating the predicates over individual tables from those related to the join conditions

Algorithm 8 Grid-AR range join query

1: **function** RANGE_JOIN_EST($Query\ Q,\ Table\ R,\ Table\ S$)
2: $\quad Q_l, Q_r, Q_j \leftarrow split(Q)$ ▷ split Q according to columns
3: $\quad GC_l \leftarrow Q_l(R)$ ▷ query estimator of R
4: $\quad GC_r \leftarrow Q_r(S)$ ▷ query estimator of S
5: $\quad card \leftarrow 0$
6: $\quad GC_{r\_card} \leftarrow cells\_card\_arr(GC_r)$ ▷ per-cell card
7: $\quad$ **for** $gc_{li} \in GC_l$ **do** ▷ for each cell qualified for Q_l
8: $\qquad op \leftarrow init\_arr(gc_{li}.card, size = |GC_r|)$
9: $\qquad$ **for** $j_c \in Q_j$ **do** ▷ for each join condition
10: $\qquad\quad GC_r \leftarrow get\_sorted(GC_r, j_c)$ ▷ sort based on j_c
11: $\qquad\quad$ **for** $gc_{rj} \in GC_r$ **do** ▷ for cells qualified for Q_r
12: $\qquad\qquad cond_{fulfill} \leftarrow check\_cond(j_c, gc_{li}, gc_{rj})$
13: $\qquad\qquad$ **if** $cond_{fulfill} == 0$ **then** ▷ unsatisfied
14: $\qquad\qquad\quad op[gc_{rj}.id] \leftarrow 0$
15: $\qquad\qquad$ **else if** $cond_{fulfill} == 1$ **then** ▷ satisfied
16: $\qquad\qquad\quad$ **break**
17: $\qquad\qquad$ **else** ▷ partially satisfied
18: $\qquad\qquad\quad op[gc_{rj}.id] *= p(j_c, gc_{li}, gc_{rj})$
19: $\qquad\qquad$ **end if**
20: $\qquad\quad$ **end for**
21: $\qquad$ **end for**
22: $\qquad card \leftarrow card + sum(op * GC_{r\_card})$
23: $\quad$ **end for**
24: $\quad$ **return** $card$
25: **end function**

(Algorithm 8, Line 2). These segmented subqueries Q_l and Q_r are then processed over the individual estimators for the respective table. Upon executing a query with a single table estimator, instead of providing a final cardinality for the entire table, we return the cardinality of the query for each grid cell (Algorithm 8, Lines 3–4). Since the per-table subqueries already filter out some of the grid cells that do not qualify for the subquery result, there will be a significantly smaller number of cells than the total number of grid cells in the estimator. Consequently, the initial prefiltering is carried out. The algorithm that operates on the individual tables is as described in Algorithm 7.

Range join query phase: The second part uses the estimates produced by Grid-AR and the grid cells produced from the individual tables to estimate the cardinality for the range join query (Algorithm 8, Lines 6–23). Let $gc_{l1}, gc_{l2}, \ldots$ be the result grid cells from the left table R and $gc_{r1}, gc_{r2}, \ldots$ be the result grid cells from the right table S. We iterate through each gc_{li}, and for each gc_{rj}, we ascertain if and to what extent the conditions specified in the range join are satisfied. The variable *op* is initialized using the cardinalities for the respective grid cell gc_{li} (Algorithm 8, Line 8). **Note that** we optimize for faster execution through early termination and parallel computation. The grid cells GC_r of S are sorted according to the join condition to allow early termination (Algorithm 8, Line 10). Since the calculations for a pair of cells are independent, we parallelize the algorithm with respect to the grid cells from table R. For each condition, if we have an expression over the values involved in the join, we apply the expression to the boundaries of the buckets for the dimension on which the expression appears. We then distinguish between *three cases*.

1 If the grid cell gc_{rj} does not satisfy at least one join predicate, we do not consider the grid cell estimate in the final cardinality (Algorithm 8, Line 13). 2 Conversely, if all of the boundaries of the grid cell gc_{rj} completely satisfy all join conditions when compared to gc_{li}, we consider a complete predicate fulfillment (Algorithm 8, Line 15). In the best case, when the per-table estimates are exactly correct, then the previous two computations will result in the correct cardinality. 3 The grid cell gc_{rj} partially satisfies the conditions specified in the range join when compared to the cell gc_{li} (Algorithm 8, Line 18). In such case between two grid cells, for each join operator θ, we calculate the probability that the r-th condition is satisfied as op_r. The same calculation is performed for all of the conditions in the join query in the end, resulting in k individual condition satisfaction probabilities $op_1, \ldots, op_k$. All probabilities are then multiplied, resulting in the overall predicate conditions probability of the whole range join query for the considered two grid cells. Intuitively, when the condition is completely not satisfied, then $op_r = 0$, and when the condition is completely satisfied, then $op_r = 1$.

The final estimate for the cardinality of two cells gc_{li} and gc_{rj} is $card_i * card_j * op_1 \ldots * op_k$, where $card_i$ and $card_j$ are the estimated cardinalities of gc_{li} and gc_{rj}, respectively. To produce the estimate, we iterate through all cells and sum over the estimated cardinality (Algorithm 8, Line 22), calculated as:

$$card = \sum_{i=1}^{n} \sum_{j=1}^{m} card_i * card_j * \prod_{r=1}^{k} op_{ijr}$$

where n and m are the number of grid cells generated from the subquery of the left table and right table, and k is the number of conditions in the range join.

Join predicate fulfillment: For the join conditions between R and S, we define the probability of a condition θ being satisfied for two grid cells as $op = p(x < y)$ or $op = p(x > y)$, i.e., the probability that a variable x is smaller (or larger) than y, where $x \in [x_{min}, x_{max}]$ and $y \in [y_{min}, y_{max}]$. The ranges correspond to the cell boundaries for the columns considered in the join condition. To calculate op, we explore several variants. One way of computation is through *sampling* from the cells. Given n samples $S_x = \{s_{x1}, \ldots, s_{xn}\}$ for x and n samples $S_y = \{s_{y1}, \ldots, s_{yn}\}$ for y, where $\forall s_{xi} : s_{xi} \in [x_{min}, x_{max}]$ and $\forall s_{yi} : s_{yi} \in [y_{min}, y_{max}]$, we calculate op as:

$$op = \frac{|S_x \, \theta \, S_y|}{n}$$

which represents the fraction of samples that satisfy the condition divided by the number of retrieved samples n. Alternatively, we can use *double integration* to determine the probability op of a condition being met. We observed that both approaches yield comparable probabilities when increasing the number of samples. Due to the computational efficiency, we chose to utilize sampling.

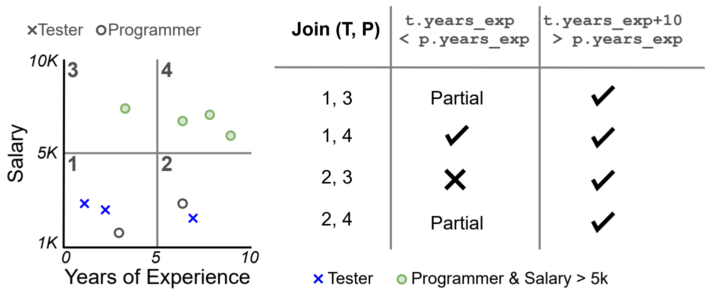


Figure 7.5: Example range join query. The grid structure with values is shown on the left-hand side and the comparison between qualifying grid cells on the right-hand side.

The complexity of a two-table join is $O(|GC_l| * |GC_r| * |Q_j|)$. The worst-case complexity occurs only when the join conditions necessitate every grid cell from one table to match every grid cell from the other table. Otherwise, the algorithm skips most cells.

Multi-Table Join Estimation

Our proposed algorithm also supports multi-table range joins. More specifically, a *chain multi-table* join is broken down into several range joins involving pairs of relations., i.e., $Q(R_1, R_2), \ldots, Q(R_{n-1}, R_n)$. The processing of the complex join query starts from the first join pair by executing Algorithm 8. However, as an output, instead of the total cardinality, the estimated cardinality is accumulated for each cell gc_j of R_2 by summing up all join cardinalities with the corresponding grid cell partners from R_1. The grid cell boundaries are adjusted in accordance with their join partners in R_1. All updated cells gc_j that have some overlap with any of the cells of R_1 together with the accumulated cardinality are used in the next join pair (R_2, R_3) as the cells of R_2. This procedure is applied for all successive join pairs, in the end producing the query cardinality.

7.4.2 Example for Range Join

Let us consider the example depicted in Figure 7.5 featuring data points from the `Employee` table. To illustrate the estimation process, we consider a self-join over the `Employee` table of the following form:

```
SELECT COUNT(*)
FROM Employee t, Employee p
WHERE t.job="Tester" AND p.job="Programmer"
AND p.salary>5000 AND t.years_exp<p.years_exp
AND t.years_exp+10>p.years_exp
```

where we join employees that are *Testers* with those that are *Programmers* with *salary* > 5000 based on years of experience. A query that involves a join between the two tables requires two Grid-AR estimators, but for this example, they will have the same structure (Figure 7.5). Initially, we evaluate the two distinct subqueries over the respective table estimator to retrieve the qualifying grid cells. Thus, for the predicate `t.job = "Tester"`, the first and the second cell qualify. For the predicates `p.job = "Programmer"` and `p.salary > 5000`, the third and the fourth cell qualify. Alongside the qualifying cell, we obtain the estimate of the cardinality for each cell given the predicates. Subsequently, we iterate through the cell combinations and assess whether each condition is satisfied. To optimize the execution, we sort the grid cells from the second table according to the dimension in the current considered condition. In this example, we will sort the cells in the order $3, 4$. For the first join condition, when comparing the grid cell 1 of the *Testers* and the grid cell 3 of the *Programmers*, we need to compute the probability of the condition being satisfied due to the overlap in their ranges. However, when comparing grid cell 1 with 4, we can ascertain that every point from cell 1 is undoubtedly smaller than any point in cell 4 since the entire range of cell 1 is smaller than that of cell 4. Thus, the probability that the condition is satisfied is $op = 1$. In contrast, the entire cell 2 is completely larger than cell 3, leading to $op = 0$, i.e., the condition is entirely not satisfied. We follow a similar process for the next join condition, where the optimization from sorting becomes salient. We can exploit the sorting order to skip the comparison between some cells. In this case, the cell order is $4, 3$. Recognizing that cell 1 is already larger than cell 4 when assessing the conditions, it implies that all subsequent grid cells in the order will also be smaller. Thus, op is set to 1 without having to compare further.

7.4.3 Practical Challenges and Observations

The estimation approach for range join queries depends on the overlap between the qualifying grid cells. When the grid cells from one relation completely (do not) satisfy the join condition, the accuracy of the range join estimate depends only on the estimate of the individual table estimators. For instance, consider two attributes c_1 and c_2 belonging to the tables R and S, respectively. For the join condition $R.c_1 < S.c_2$, when comparing two cells from each estimator, if we know the range of the dimensions c_1 and c_2 in both cells and we know that $c_{1max} < c_{2min}$, then the true cardinality is the multiplication of the cardinality of the individual cells, as every point in c_1 is definitely smaller than every point in c_2. However, since we rely on per-cell estimates of the individual estimators, even when the join condition is fully satisfied between the considered cells, *a poor estimate of the estimators will contribute to worse estimation accuracy over range join queries.* The absolute error of this part is $\sum_{gc_i \in R_1} \sum_{gc_j \in R_2} |(card_{gc_i} * card_{gc_j}) - (est_{gc_i} * est_{gc_j})|$. Thus, a better accuracy in the estimator leads to more accurate join estimation. For grid cells that partially satisfy the conditions, to produce representative samples, we need to *capture the per-column data distribution.* Although this can improve the accuracy, it increases the estimator size and the time required for training the models. From our evaluation, capturing the distribution leads to a slight improvement over always assuming uniformity while drastically increasing the creation time.

Related work [166] relies on precomputing the actual equi join or its statistics to achieve accurate estimates. Given the numerous combinations, it is <u>not feasible</u> to materialize every possible range join with all potential expressions unless the join is predetermined before query estimation. Also optimizing an estimator for a single specific join variant is rarely practical. Our algorithm avoids pre-

Table 7.1: Datasets and query characteristics.

Datasets	**Customer**	**Flight**	**Payment**
Number of Rows	$n = 150k$	$n = 2.1M$	$n = 8.8M$
Number of Cols.	8	9	7
Textual / Numerical Cols.	5 / 3	3 / 6	3 / 4
Max Query Predicates	5	7	5
Max Join Tables	5	5	/

computing the join and can handle any range predicate and expression. By partitioning the space, large portions of rows can be dismissed or, respectively, taken into account, which leads to accurate estimates.

7.5 Experimental Evaluation

We have realized our approach in Python. The CDF model has been implemented using SKlearn, and the autoregressive model using PyTorch. The experiments for the learned models, i.e., the supervised model for CDF estimation and the autoregressive model, are carried out on an NVidia GeForce RTX 2080 Ti GPU. The computations of the grid and the competing non-learned approaches are executed on an Intel Xeon E5-2603v4 CPU@1.7GHz, 128GB RAM.

The columns with textual values and dates are encoded using a dictionary prior to evaluation. For Grid-AR, the columns with continuous data are directly used to create the grid. To make the competing autoregressive model handle floating values, such columns are encoded using a dictionary before the evaluation. The approach represents integer attributes with 32-bits. When comparing the autoregressive model used in Grid-AR to the standalone autoregressive model, we use three *layers* with 512 neurons and train the model for ten *epochs*. We use an *embedding size* of 32 for the input and output. We use per-column *compression* [39] for columns with more than 2000 unique values by splitting the column into two subcolumns. The autoregressive model of Grid-AR always uses compression. The per-column CDF model is a Decision Tree Regressor where we varied the depth between 3 and 6. Naturally, higher number of layers produces more accurate results while requiring more time for the computation.

For single table queries, we report on the **accuracy**, **memory**, **training time**, and **estimation time**. For range joins, we report the *accuracy* and *estimation time* using the same Grid-AR structures as for the single table estimates. For measuring the accuracy, we use $q{-}error = max(\frac{true(q)}{estimate(q)}, \frac{estimate(q)}{true(q)})$.

7.5.1 Datasets

We evaluated the approaches on one synthetic and two real-world datasets. The characteristics of the datasets are detailed in Table 7.1.

- **TPC-H:** We use the **Customer** table from the *TPC-H* synthetic benchmark with scaling factor one. The table has only one float attribute, which for standalone autoregressive models is dictionary mapped.

- **Flight:** The real-world dataset **Flight** has flight sensor data from Germany. The sensors produce data containing nine attributes, where all numerical attributes are of floating point type. Thus, these floating point attributes need a dictionary mapping for the autoregressive model.
- **Payment:** This real-world dataset contains information about the billing of a mid-sized company. We have selected the **Payment** relation with information about payments of bills issued to customers, such as the amount, date and type of payment, and others. Of the seven attributes, two have floating point values and need additional dictionary mapping.

For single table queries, we generated 300 queries for *Customer* and 200 queries for *Flight* and *Payment* with **varying numbers of query predicates** where the maximum number of predicates per dataset is specified in Table 7.1. The predicates have one of the operations: $=, >, <, \leq, \geq$, where for inequalities, on the same attribute, both an upper and lower bound can be specified. For **range join queries**, we used the tables **Customer** and **Flight** and created range join queries by performing self-joins for different join types and number of joins. For every table, we created 150 range join queries with a varying number of equality and range predicates as for the single table queries. For both datasets, a *multi-join query can include up to five tables.* Between two tables, for **Customer**, the queries have up to three join conditions and, for **Flight**, up to five. We group the range join types into **inequality**, i.e., inequality comparison of two attributes, and **range**, i.e., a point in interval or interval overlap join. We do not consider group-by or nested queries.

7.5.2 Competitors

In this chapter, we consider the following approaches in our experiments.

Queries over Single Tables

- Our hybrid approach **Grid-AR**. By default Grid-AR, uses a CDF–based grid. We study the impact of uniform vs. CDF–based grid and grid resolution later-on separately.
- Since our approach aims to improve the autoregressive models by overcoming their problems when working with numerical and textual attributes, as our main competitor, we use the *autoregressive model* architecture named Naru [167]. We use the code provided by the authors which includes progressive sampling to estimate queries with range predicates. The model is trained over all rows of the table by first performing dictionary encoding for the attributes with floating point and textual values. However, since highly heterogeneous attributes directly affect the model size and estimation time, we pair the model with a per-column compression technique [39]. To avoid showing the impact of the different per-element compressions, instead of comparing to NeuroCard [166], we apply the compression used by Grid-AR to Naru. Thus, we consider two versions of Naru as competitors: one without compression, **Naru**, and the other with compression, **CNaru**. The use of compression dictates a change in the progressive sampling estimation algorithm. First, the values of the query predicates are compressed before they are sent to the model. Second, in the estimation

algorithm, for a compressed column, the possible values for every next subcolumn are restricted by the values from the previous subcolumn(s). For *Payment*, we could not fit *Naru* on our GPU even after drastically reducing the batch and embedding size. Thus, we do not present results for the estimation time and accuracy for this dataset.

- We also consider **IAM** [114], which is a modification on top of Neurocard [166] that integrates Gaussian mixture models (GMM) with deep autoregressive models to separate the handling of continuous and categorical attributes. We use the publicly available implementation with the same parameter settings for the number of layers (4) and units (256, 128, 128, 256), training epochs (10), number of GMM components for each attribute (30), samples for training (10^6), progressive sampling, and GMM, and number of subcolumns (2) as proposed by the authors. To depict the memory consumption, we use the model with the smallest embedding size.
- To outline the effects of our approach, as a baseline for producing exact answers, we include PostgreSQL 14 (**Postgres**). We also include the results of the classical assumption techniques paired with per-column 1D histograms of PostgreSQL (**EPostgres**).

Range Join Queries

To the best of our knowledge, this is the first approach that addresses cardinality estimation over range joins. Thus, as baseline for exact answers, instead of PostgreSQL, we compare to the state-of-the-art solutions for exact answers, namely, **DuckDB** v.0.9.2 and **Tableau Hyper** v.0.0.13129, both used as Python packages. Both solutions offer faster execution, drastically **outperforming** PostgreSQL algorithms for range join queries. We further compare to the estimation approach included in PostgreSQL 14 (**EPostgres**).

7.5.3 Cardinality Estimation over Single Tables

Accuracy

In the following, we present the accuracy results of our proposed approach compared to the considered competitors. The accuracy results (Table 7.2) are an average of *five* executions. To represent the accuracy, we show the average, median, 90th percentile, and maximal q-error for the considered datasets. A higher q-error means worse estimation accuracy.

When analyzing the results for all datasets, it is evident that Grid-AR produces comparable results or slightly less accurate when compared to the best approach, IAM, in terms of median accuracy. However, for *Payment*, Grid-AR drastically outperforms the competitors when considering the average q-error. For the *Flight* dataset, Grid-AR produces estimates with the best median q-error while having a comparable average q-error to IAM. For the higher percentiles, CNaru has worse estimation accuracy than Naru. The reason for this is the introduced compression, which affects the correlation between the columns, introducing more complex interconnections for the model to learn. This also applies to Grid-AR since we always use the compressed version of the autoregressive model to make it applicable to highly heterogeneous datasets.

Table 7.2: Estimation accuracy expressed as q-error.

Datasets	**Payment**				**Flight**				**Customer**			
Approach	**Median**	**90th**	**Max**	**Average**	**Median**	**90th**	**Max**	**Average**	**Median**	**90th**	**Max**	**Average**
EPostgres	2.5	21.7	$1.7 * 10^5$	3075.5	2.4	7.5	115	5.2	2.4	2.5	3.5	2.4
Naru	*n/a*	*n/a*	*n/a*	*n/a*	1.5	9.5	179.5	7.6	**1.07**	1.4	**2.7**	1.2
CNaru	1.2	**4**	$3 * 10^4$	540.8	1.5	10	298	11.1	**1.04**	**1.2**	5.2	**1.1**
IAM	**1.1**	13.7	6308.5	116.3	1.3	**6**	59.5	**3.1**	**1.09**	1.6	1366	12.5
Grid-AR	1.38	32	**496**	**20.1**	**1.25**	7	**59**	3.6	**1.09**	1.4	7.8	1.2

Table 7.3: Training time in seconds represented as the average time for training per epoch.

Datasets	**Payment**	**Flight**	**Customer**
Approach	**Average Time**	**Average Time**	**Average Time**
Naru	n/a	2600	39.1
CNaru	136.1	39.2	4.1
IAM	270.8	744.6	36.1
Grid-AR	**85.5**	**24**	**3.2**

Table 7.4: Estimation time in milliseconds represented as the median and average time per query.

Datasets	**Payment**		**Flight**		**Customer**	
Approach	**Average**	**Median**	**Average**	**Median**	**Average**	**Median**
Postgres	553.4	539	232.1	223.8	20.2	19.3
EPostgres	**0.37**	**0.29**	**0.33**	**0.31**	**0.2**	**0.18**
Naru	*n/a*	*n/a*	509.6	527.6	62.6	82.1
CNaru	17	8.5	459.1	489.8	55.2	72.9
IAM	15.9	12.7	15.6	15.2	19	18.7
Grid-AR	14.4	7.4	8.7	6.8	1.5	1.5

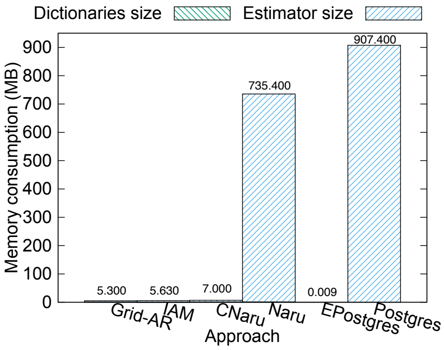
EPostgres produces the worst estimation accuracy, visible for highly heterogeneous datasets such as *Payment*, producing substandard results that cannot be considered in any real-world scenario requiring cardinality estimation. When analyzing the accuracy while considering the estimation time and memory, it is clear that our approach meets the expectation of achieving comparable accuracy while producing the results much faster and occupying drastically less memory.

Training and Estimation Time

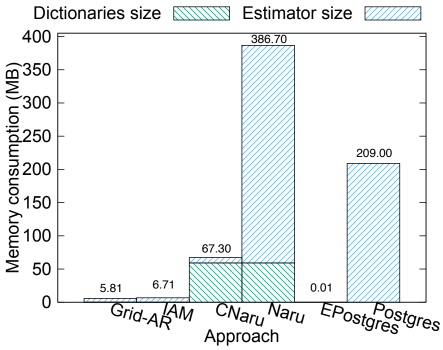
Next, we detail the training time for the learned approaches requiring training, and the estimation time for all approaches for the three datasets. When considering *Naru* and *CNaru*, for the progressive sampling-based estimation, we always select 1000 samples although the authors propose the selection of 2000 samples, we found that the estimation time drastically increases for such setting while the improvement in accuracy is not notable to justify the increased estimation time. Therefore, for our experiments, we always select 1000 samples. All autoregressive models have our standard setting of three layers with 512 neurons and embedding of the input and output with size 32.

The training time results (Table 7.3) underpin the importance of compression for highly heterogeneous datasets, considering not only the memory but also the training efficiency. Going a step beyond, our Grid-AR approach groups points into buckets, yielding a smaller volume of training data. Grid-AR requires *less training time* while managing to produce estimates *faster*. For all considered datasets, Grid-AR consistently achieves the best training time.

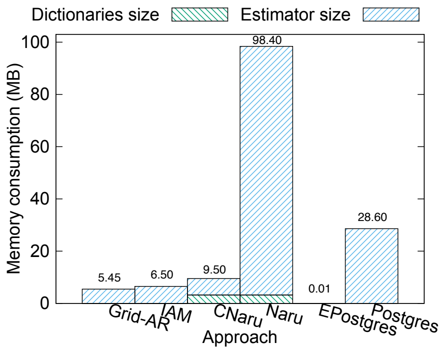
When analyzing the estimation times presented in Table 7.4, it is clear that Grid-AR *outperforms all the considered approaches except for EPostgres*, where



(a) Payment



(b) Flight



(c) Customer

Figure 7.6: Memory consumption in megabytes considering the size of both the estimator and the dictionary mapping.

in most scenarios, it produces results that are orders of magnitude better than those of the competitors. When considering the average estimation time, we can conclude that Grid-AR can produce the estimates blazingly fast, where, for some datasets, it even contends the estimation time of per-column 1D histograms that are used by EPostgres. When analyzing the results for the *Customer* and *Flight* datasets, the difference in the estimation time to the competitors is especially prominent. The main reason for this is the presence of more numerical attributes. For Grid-AR, these attributes will be included in the grid, whereas for Naru and CNaru, proper value mapping and optional compression will be performed. Due to the large number of numerical columns, the competitors have an increased number of attributes in the model, directly influencing the iterative sampling required for the range queries, plus an additional overhead of finding a proper integer mapping for the floating values. Although the input will be smaller for IAM, the iterative sampling algorithm will still need to be performed. Grid-AR avoids the costly operations of mapping and iterative sampling required by the competitors, drastically outperforming them. Producing estimates fast is crucial for any estimation approach, so these results further show the importance of our solution of splitting the estimation to consider numerical and text-valued columns independently.

Table 7.5: Accuracy (q-error) of estimator variants when varying the total number of grid cells (Payment).

Grid cells	**10 000**		**20 000**		**30 000**		**50 000**	
Approach	**Median**	**Avg**	**Median**	**Avg**	**Median**	**Avg**	**Median**	**Avg**
Uniform	1.2	459	1.1	417	1.1	409	1.1	375
CDF	1.4	84	2	22	1.3	20	1.6	14

Memory Consumption

We next detail the memory consumption of Grid-AR compared to the considered competitors. When measuring the memory, we consider two parts: the memory of the estimator and the memory of the value-integer mappings. For Grid-AR, we store the grid structure, autoregressive model, and the mappings for the textual columns. For the autoregressive models, we store the mappings of the textual and floating-point columns in addition to the model. For Postgres, we consider table size, and for EPostgres, the sum of the sizes of the per-column 1D histograms. The required memory, in megabytes, is shown in Figure 7.6. When analyzing the results, it is clear that after EPostgres, Grid-AR occupies the least memory for all datasets. This difference is especially noticeable for the *Flight* dataset (Figure 7.6b) due to the number of columns that have floating-point values. Since more than half the columns are of this type, Naru and CNaru need to store additional mapping for them, as shown by the *Dictionary Size* value. For the *Payment* dataset (Figure 7.6a), since there is only one column that has floating-point values, the difference between the required memory of Grid-AR and CNaru is smaller. Nevertheless, Grid-AR requires 25 percent less memory than CNaru. For IAM, the usage of Gaussian mixture models requires slightly more memory than our Grid-AR approach.

Grid-AR has the best balance among all approaches by requiring the least memory and producing results with comparable accuracy to the most accurate approach but in a drastically faster time.

7.5.4 Grid-AR Analysis

In addition to the comparison to the competitors, to showcase the importance of correctly capturing the per column distributions, we inspect and compare the two versions of Grid-AR based on how we split the ranges per dimension in the grid. In these experiments, the version that uses uniformity assumption is called *Grid-AR Uniform*, and the version using per dimension CDFs for creating the cells is called *Grid-AR CDF*. We compare the versions by varying the number of grid cells and present results for the estimation accuracy, memory consumption, and estimation time.

Accuracy

When analyzing the results in Table 7.5, we notice that as the size of the grid increases, as expected, we observe an improvement in the estimation accuracy. For both versions, this improvement is especially prominent in the average accuracy. However, the median accuracy, even for the smallest grid, is already of

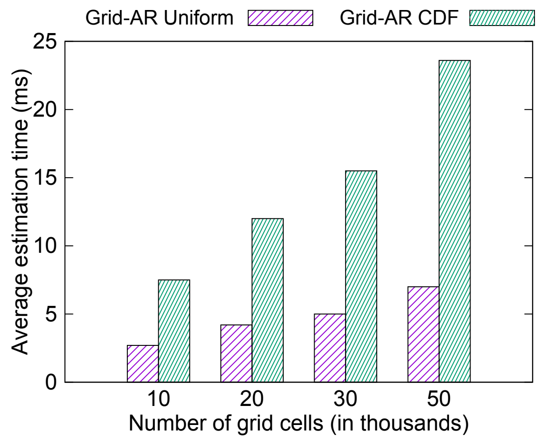


Figure 7.7: Estimation time in milliseconds when varying the total number of grid cells (Payment).

acceptable quality and comparable to the largest grid structure. Grid-AR CDF has better average accuracy for all considered grid sizes, where starting from the smallest grid (10000 cells), it drastically outperforms Grid-AR Uniform. These results outline the importance of correctly capturing the distribution of highly heterogeneous attributes and the effect that this has on the results.

Estimation Time

To analyze the estimation time of the two grid versions, we depict the median estimation time in Figure 7.7. The increase in the number of cells of the grid structure directly contributes to increased estimation time (Figure 7.7). However, although there is a sharp increase in the number of cells, the estimation time is still in the range of milliseconds. In contrast to Grid-AR Uniform, Grid-AR CDF requires additional computation of the CDF to gather the qualifying cells for a user-imposed query. However, the additional overhead is not significant, especially when considering the improvement in accuracy (Table 7.5).

Memory Consumption

The memory consumption of the grid structures when varying their number of cells is depicted in Figure 7.8. For both versions, as the number of cells increases, there is almost no difference in the memory consumption of the grid structures. The constituent autoregressive model occupies the largest portion of the memory for both versions. Despite the increase in the number of cells for each version, this difference, although noticeable, will not be highly pronounced due to our use of per-element compression for the remaining columns. In addition, the grid stores the bucket boundaries for the numerical attributes of the relation. Thus, as the number of cells increases, so does the number of bucket boundaries for the attributes, resulting in increased information that needs to be stored. When directly comparing the two versions, Grid-AR CDF has slightly larger memory than Grid-AR Uniform due to the additional overhead from storing a regressor per dimension.

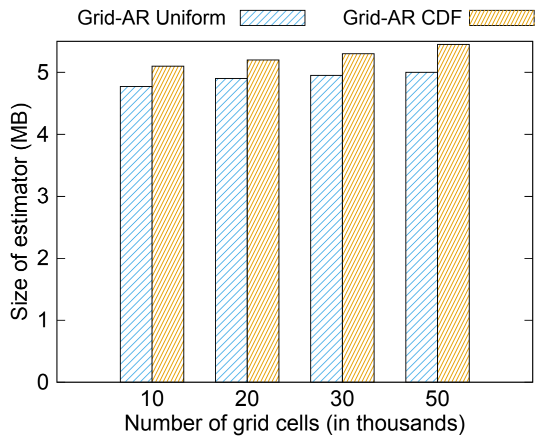


Figure 7.8: Memory consumption in megabytes when varying the total number of grid cells (Payment).

Table 7.6: Accuracy (q-error) for range join queries over two tables for different join types on Customer and Flight.

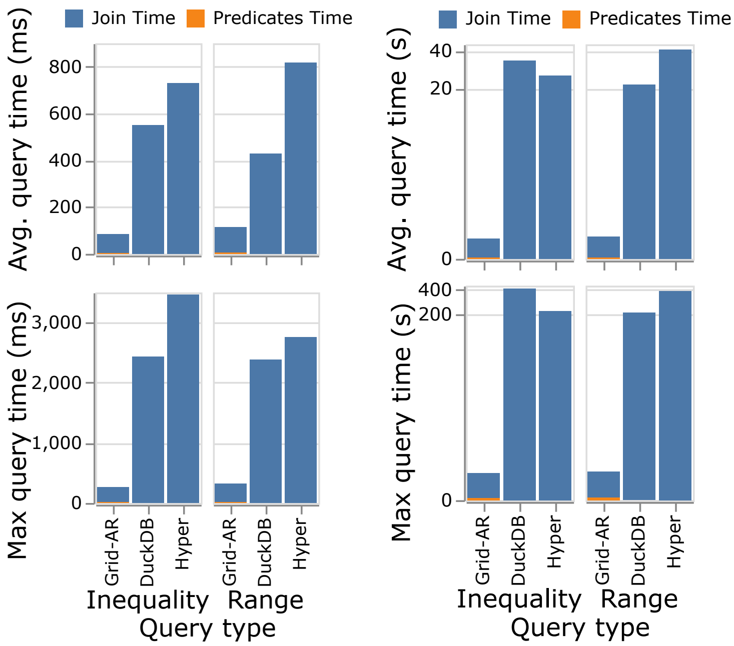
Approach	**Query Type**	**Inequality**		**Range**	
	Dataset	**Median**	**Average**	**Median**	**Average**
EPostgres	Flight	10	169.5	21.8	303.1
	Customer	3.85	7	5.1	8.5
Grid-AR	Flight	**1.44**	**5.9**	**1.56**	**11.2**
	Customer	**1.25**	**4.1**	**1.2**	**1.8**

7.5.5 Cardinality Estimation for Range Joins

We report the *estimation time* of all approaches and the *accuracy* of our approach and EPostgres. We use the same Grid-AR per dataset as for the single table query evaluation. There is one Grid-AR for every table in the range join query. The memory of our approach, per table, remains the same as for the previous evaluation. The competitors do not store additional indexes. From the 150 queries for every dataset, the number of inequality and range join queries is proportional.

Accuracy

The estimation accuracy grouped by the different query types is depicted in Table 7.6. From the results, it is evident that Grid-AR *drastically outperforms* EPostgres. For Flight, the inequality joins achieve better overall accuracy than the range joins. This is because in range join queries, comparing cells from the estimators of the different tables leads to more cells that partially overlap with one another. This, in turn, triggers the computation of the overlap estimation of the cells, further affecting the per-cell estimated cardinalities. Note that the estimates for the range join queries (inequality and range) directly depend on the quality of the individual Grid-AR structures. If the underlying structure



(a) Customer (b) Flight

Figure 7.9: Average and maximal estimation time per query type for range joins over two tables on **Customer** (left) and **Flight** (right). For Flight, y-axis is log scaled.

produces inferior estimates for the predicates over the individual tables, their accuracy further deteriorates when capturing the range join.

Estimation Time

We present the average and maximal estimation time per query type in Figure 7.9. For Grid-AR, we split the time into *predicates time*, which is the time that the per-table estimators need for estimating the qualifying cells for the query and their cardinality, and *join time*, which is the time for computing the join result from the qualifying cells. From the shown results, it is evident that Grid-AR has the best estimation time for both datasets. When considering the *Flight* dataset (note the log scale), whose table is significantly larger than the *Customer* table, Grid-AR *produces estimates in milliseconds* where the competitors require, on average, more than 20 seconds. When analyzing the maximal time, it is clear that Grid-AR performs efficiently even for not-so-selective queries. *For EPostgres, the estimation time is between* 0.5 and 1 ms, irrespective of the query type or cardinality.

Multi-table Range Joins

To showcase the generalizability of our range join approach, we created join queries that involve three, four, and five tables, consisting of only inequality

Table 7.7: Median accuracy (q-error) for range join queries with three, four, and five tables on Customer and Flight.

Approach	**Query Type**	**Inequality**			**Range**			**Combination**		
	Dataset	**3**	**4**	**5**	**3**	**4**	**5**	**3**	**4**	**5**
EPostgres	Flight	10.5	147.9	71.1	15.2	264.7	16.3	9.1	36.4	153
	Customer	2.4	5.9	4.9	14.1	63.9	29.9	5.8	4	6
Grid-AR	Flight	7.8	7.8	29.5	9	24	11.9	3.9	13	45
	Customer	1.5	1.8	1.7	1.8	2.4	2.3	1.6	1.7	1.8

Table 7.8: End-to-end time for range join queries (Flight).

Approach	**End-to-end Time**		**Improvement**	
	Median	**Average**	**Median**	**Average**
EPostgres	2s	9.35s	–	–
Exact Card. (optimal)	1.43s	8.23s	28.5%	11.98%
Grid-AR	1.44s	8.28s	28%	11.44%

joins, only range joins, and a combination of both. The accuracy results expressed through the q-error are shown in Table 7.7. Grid-AR continues to outperform EPostgres in terms of estimation accuracy for the different join queries and number of involved tables. For the uniformly distributed *Customer* dataset, Grid-AR produces estimates that are almost always at least twice as good as those of EPostgres. EPostgres took the same average time as for the queries with two tables. For all join types and number of tables, Grid-AR took on average 600 milliseconds for *Flight* and 21 milliseconds for *Customer*. For DuckDB, the average time was 11 seconds for *Flight* and 116 seconds for *Customer*. For Hyper, the average time was 32 seconds for *Flight* and 71 seconds for *Customer*.

7.5.6 Overall Performance

To examine the effects of Grid-AR on query optimization concerning the end-to-end query execution, we integrated our approach into PostgreSQL 14 following the procedure from related work [25,114]. We ingest the cardinality estimates in the query optimizer of PostgreSQL such that for every query we incorporate the estimates of the subqueries. The modified PostgreSQL then uses the estimates to generate a query plan for which we measure the end-to-end query execution time. We used 300 queries that consist of range joins, inequality joins, or a combination of both, having up to five joining tables, and compared Grid-AR to the native PostgreSQL estimation and having the exact cardinalities as estimates. The results are shown in Table 7.8. We measure the end-to-end query execution time, planning time, and the improvement (percentage) over the native Postgres estimation. The improvement is calculated as (time of EPostgres - approach time) / time of EPostgres.

As depicted, the improvements in the query runtime are prominent, especially for the median query execution time. Grid-AR has almost the same performance as the optimal approach with exact query cardinalities. These results further support the claims gathered from the elaborate evaluation of range join queries. Thus, our empirical evaluation demonstrates the importance of accurate cardinality estimates, especially when considering the unexplored field of range joins.

7.6 Summary

In this chapter, we proposed Grid-AR, a novel, hybrid cardinality estimator. By combining a grid structure with autoregressive models, Grid-AR distinguishes between column types, separating the continuous from categorical ones. Grid-AR helps circumvent two intrinsic problems of autoregressive cardinality estimators. Through partitioning in the grid, we avoid storing large dictionary mappings of the numerical columns, greatly diminishing the space needed for the estimator. To reduce the search space and bypass the slow iterative sampling, we proposed a faster batch algorithm for range queries that leverages the grid. The experimental evaluation showed that Grid-AR exceeds the state-of-the-art approaches regarding memory and speed while achieving comparable accuracy when applied over single tables. We further demonstrated the importance of a cardinality estimator for range join queries that can promptly deliver accurate estimates.

Chapter 8

Conclusion and Outlook

The goal of this thesis was to explore and propose algorithms for exact query processing and approximate estimation using data summaries. Our research focused on both query processing over vast static datasets and continuously generated data, specifically data streams. In particular, we addressed partitioning and joining of schema-free data, as well as data summarization in a distributed streaming context. Recognizing the need for methods that do not require processing the entire dataset, we studied approximate query processing and cardinality estimation when working with static data.

We tackled the challenge of computing natural joins on schema-free JSON documents. We reached our goal by splitting the computation into two tasks. First, we identified an effective data partitioning schema to group the data, and second, we proposed algorithms to ensure efficient local query evaluation. Our framework introduces novel solutions for both tasks. To guide the partition creation, we leveraged association analysis principles. For join computation, we employed FP-trees, which provide a compact representation of JSON documents. Building on this, we developed two variants of a join algorithm that utilize the FP-tree structure to effectively prune non-relevant candidate pairs, optimizing local execution across compute nodes. To address scalability challenges caused by attributes with low-value variety, we implemented a countermeasure to maintain efficiency. Our performance evaluation demonstrated the approach's resource efficiency and capacity to manage large data volumes effectively. Additionally, our experimental evaluation highlighted the substantial performance improvements of our join algorithms over traditional join algorithms, underscoring the practicality and efficiency of our algorithms.

We also explored the problem of continuously extracting summaries from data streams, by making use of core items when working in an edge computing setting. In this context, generating summaries can drastically reduce network load and speed up the further processing. We introduced a novel algorithm, SoftSieving, for stream summarization based on core items. SoftSieving employs a small, fixed number of dynamic sieves that support parallel processing. The algorithm produces "soft" summaries, allowing core items to be replaced by others with greater utility gains. Our experimental evaluation demonstrated that SoftSieving achieves a favorable balance between processing speed and utility, outperforming traditional sieve-based algorithms.

For extensive static datasets, we tackled the challenge of approximate query processing by shifting from traditional per-tuple processing to an approach that operates on compact representations of tuple groups, which we term tuple bubbles. To create concise yet accurate representations, we proposed using Bayesian networks and autoregressive models, evaluating their effectiveness in compactly encoding tuple bubbles while supporting queries with any combination of equality joins and equality or range predicates. We outlined key challenges and validated our approach with a preliminary evaluation, which demonstrated notable improvements in accuracy and query estimation times. Thus, we motivated the move from tuple-level to bubble-level processing and established the foundation by concentrating on approximate query processing.

Using autoregressive models for approximate query processing, we have identified opportunities for their optimization in the context of cardinality estimation. Consequently, we proposed our approach, Grid-AR, for cardinality estimation over single tables. Grid-AR combines a grid structure with an autoregressive model with the goal of distinguishing between continuous and categorical column types. This design allowed us to overcome two core challenges associated with autoregressive cardinality estimators. By using grid-based partitioning, Grid-AR eliminates the need for large dictionary mappings of numerical columns, significantly reducing the memory footprint. To further enhance efficiency, we developed a faster batch algorithm for range queries that leverages the grid to minimize the search space and avoid slow, iterative sampling. Experimental results demonstrated that Grid-AR outperforms current approaches in both memory usage and processing speed, while delivering comparable accuracy.

8.1 Outlook

During the development of our algorithms for join processing in a distributed streaming environment, our focus was mainly on self-joins over the data streams. Although we investigated the joining beyond self-joins and showed the suitability of our proposed solutions, we still need to address all the intrinsic aspects that come with join processing. In particular, we can focus on optimizing the join order when considering more than two sources. Currently, we order the joins by always selecting the most selective tree as the next one to be processed. However, a cost model that considers not only individual trees but also can estimate the potential join result will be of direct benefit to our approach. In addition, at every compute node, the FP-trees are being recomputed for every window. Since this directly affects the execution time, one approach to overcome this can be to monitor the arriving tuples and perform periodical updates of FP-trees by using delta updates [143].

Considering our work on approximate query processing using bubbles for their creation, we only examined horizontal partitioning of the relations. We can go beyond and evaluate various partitioning methods that also take into consideration dependencies between the attributes. Furthermore, although our work was on static data, the idea of moving the processing from tuples to bubbles can be highly beneficial in a distributed environment when handling data that cannot be efficiently processed on a single machine or if data shipping is prohibitively expensive. Investigating such an environment should be complementary to the analysis of possible partitioning algorithms.

List of Figures

List of Algorithms

List of Tables

Bibliography

[1] Apache zookeeper. `https://zookeeper.apache.org/`, 2008. Accessed: 2024-10-21.

[2] Apache storm. `https://storm.apache.org/`, 2011. Accessed: 2024-10-15.

[3] JavaScript Object Notation. `https://www.json.org/`, 2013. Accessed: 2024-10-15.

[4] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, pages 181–192. ACM Press, 1999.

[5] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In I. Manolescu, S. Spaccapietra, J. Teubner, M. Kitsuregawa, A. Léger, F. Naumann, A. Ailamaki, and F. Özcan, editors, *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*, volume 426 of *ACM International Conference Proceeding Series*, pages 99–110. ACM, 2010.

[6] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In Hanzálek et al. [72], pages 29–42.

[7] C. C. Aggarwal. On biased reservoir sampling in the presence of stream evolution. In U. Dayal, K. Whang, D. B. Lomet, G. Alonso, G. M. Lohman, M. L. Kersten, S. K. Cha, and Y. Kim, editors, *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 607–618. ACM, 2006.

[8] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. Nodb: efficient query execution on raw data files. In K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 241–252. ACM, 2012.

[9] M. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *Proc. VLDB Endow.*, 5(10):1064–1075, 2012.

[10] F. Alvanaki and S. Michel. Tracking set correlations at large scale. In C. E. Dyreson, F. Li, and M. T. Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1507–1518. ACM, 2014.

[11] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman. Photon: fault-tolerant and scalable joining of continuous data streams. In K. A. Ross, D. Srivastava, and D. Papadias, editors, *Proceedings of the ACM*

SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013, pages 577–588. ACM, 2013.

[12] A. Badanidiyuru, B. Mirzasoleiman, A. Karbasi, and A. Krause. Streaming submodular maximization: massive data summarization on the fly. In S. A. Macskassy, C. Perlich, J. Leskovec, W. Wang, and R. Ghani, editors, *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*, pages 671–680. ACM, 2014.

[13] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on modern processor architectures. *IEEE Trans. Knowl. Data Eng.*, 27(7):1754–1766, 2015.

[14] R. Barber, G. M. Lohman, I. Pandis, V. Raman, R. Sidle, G. K. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. Memory-efficient hash joins. *Proc. VLDB Endow.*, 8(4):353–364, 2014.

[15] K. S. Beyer, P. J. Haas, B. Reinwald, Y. Sismanis, and R. Gemulla. On synopses for distinct-value estimation under multiset operations. In C. Y. Chan, B. C. Ooi, and A. Zhou, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 199–210. ACM, 2007.

[16] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in mapreduce. In A. K. Elmagarmid and D. Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 975–986. ACM, 2010.

[17] P. Bodik, W. Hong, C. Guestrin, S. Madden, M. Paskin, and R. Thibaux. Intel wireless dataset, 2004. http://db.csail.mit.edu/labdata/labdata.html.

[18] P. Bourhis, J. L. Reutter, and D. Vrgoc. JSON: data model and query languages. *Inf. Syst.*, 89:101478, 2020.

[19] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984.

[20] G. Brown, A. C. Pocock, M. Zhao, and M. Luján. Conditional likelihood maximisation: A unifying framework for information theoretic feature selection. *J. Mach. Learn. Res.*, 13:27–66, 2012.

[21] N. Bruno, S. Chaudhuri, and L. Gravano. Stholes: A multidimensional workload-aware histogram. In S. Mehrotra and T. K. Sellis, editors, *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 211–222. ACM, 2001.

[22] N. Buchbinder, M. Feldman, and R. Schwartz. Online submodular maximization with preemption. In P. Indyk, editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 1202–1216. SIAM, 2015.

[23] S. Buschjäger, P. Honysz, L. Pfahler, and K. Morik. Very fast streaming submodular function maximization. In N. Oliver, F. Pérez-Cruz, S. Kramer, J. Read, and J. A. Lozano, editors, *Machine Learning and Knowledge Discovery in Databases. Research Track - European Conference, ECML PKDD 2021, Bilbao, Spain, September 13-17, 2021, Proceedings, Part III*, volume 12977 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 2021.

[24] S. Buschjäger, K. Morik, and M. Schmidt. Summary extraction on data streams in embedded systems. In M. Sayed Mouchaweh, A. Bifet, H. Bouchachia, J. Gama, and R. P. Ribeiro, editors, *Proceedings of the Workshop on IoT Large Scale Learning from Data Streams co-located with the 2017 European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML-PKDD 2017), Skopje, Macedonia, September 18-22, 2017*, volume 1958 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2017.

[25] W. Cai, M. Balazinska, and D. Suciu. Pessimistic cardinality estimation: Tighter upper bounds for intermediate join cardinalities. In P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 18–35. ACM, 2019.

[26] A. Chakraborty and A. Singh. Parallelizing windowed stream joins in a shared-nothing cluster. *CoRR*, abs/1307.6574, 2013.

[27] P. Chardaire, M. Barake, and G. P. McKeown. A probe-based heuristic for graph partitioning. *IEEE Trans. Computers*, 56(12):1707–1720, 2007.

[28] C. Chasseur, Y. Li, and J. M. Patel. Enabling JSON document stores in relational systems. In A. Bonifati and C. Yu, editors, *Proceedings of the 16th International Workshop on the Web and Databases 2013, WebDB 2013, New York, NY, USA, June 23, 2013*, pages 1–6, 2013.

[29] S. Chaudhuri, G. Das, and V. R. Narasayya. A robust, optimization-based approach for approximate answering of aggregate queries. In S. Mehrotra and T. K. Sellis, editors, *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 295–306. ACM, 2001.

[30] S. Chaudhuri, B. Ding, and S. Kandula. Approximate query processing: No silver bullet. In S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 511–519. ACM, 2017.

[31] J. Chen and Q. Zhang. Distinct sampling on streaming data with near-duplicates. In J. V. den Bussche and M. Arenas, editors, *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, pages 369–382. ACM, 2018.

[32] Y. Chen and K. Yi. Two-level sampling for join size estimation. In S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 759–774. ACM, 2017.

[33] C. K. Chow and C. N. Liu. Approximating discrete probability distributions with dependence trees. *IEEE Transactions on Information Theory*, 14(3):462–467, 1968.

[34] G. F. Cooper. The computational complexity of probabilistic inference using bayesian belief networks. *Artif. Intell.*, 42(2-3):393–405, 1990.

[35] G. Cormode, M. N. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Found. Trends Databases*, 4(1-3):1–294, 2012.

[36] G. Das, C. M. Jermaine, and P. A. Bernstein, editors. *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018.* ACM, 2018.

[37] A. Dasgupta, R. Kumar, and S. Ravi. Summarization through submodularity and dispersion. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, ACL 2013, 4-9 August 2013, Sofia, Bulgaria, Volume 1: Long Papers*, pages 1014–1022. The Association for Computer Linguistics, 2013.

[38] A. Davitkova, D. Gjurovski, and S. Michel. Compressing (multidimensional) learned bloom filters. In *Workshop on Databases and AI*, 2021.

[39] A. Davitkova, D. Gjurovski, and S. Michel. LMKG: learned models for cardinality estimation in knowledge graphs. In J. Stoyanovich, J. Teubner, P. Guagliardo, M. Nikolic, A. Pieris, J. Mühlig, F. Özcan, S. Schelter, H. V. Jagadish, and M. Zhang, editors, *Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022*, pages 2:169–2:182. OpenProceedings.org, 2022.

[40] A. Deshpande, M. N. Garofalakis, and R. Rastogi. Independence is good: Dependency-based histogram synopses for high-dimensional data. In S. Mehrotra and T. K. Sellis, editors, *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 199–210. ACM, 2001.

[41] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In B. Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*, pages 1–8. ACM Press, 1984.

[42] B. Ding, S. Huang, S. Chaudhuri, K. Chakrabarti, and C. Wang. Sample + seek: Approximating aggregates with distribution precision guarantee. In F. Özcan, G. Koutrika, and S. Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference*

2016, San Francisco, CA, USA, June 26 - July 01, 2016, pages 679–694. ACM, 2016.

[43] C. Durkan and C. Nash. Autoregressive energy machines. In K. Chaudhuri and R. Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 1735–1744. PMLR, 2019.

[44] D. Durner, V. Leis, and T. Neumann. JSON tiles: Fast analytics on semi-structured data. In Li et al. [100], pages 445–458.

[45] A. Dutt, C. Wang, A. Nazi, S. Kandula, V. R. Narasayya, and S. Chaudhuri. Selectivity estimation for range predicates using lightweight models. *Proc. VLDB Endow.*, 12(9):1044–1057, 2019.

[46] M. Elseidy, A. Elguindy, A. Vitorovic, and C. Koch. Scalable and adaptive online joins. *Proc. VLDB Endow.*, 7(6):441–452, 2014.

[47] C. Estan and J. F. Naughton. End-biased samples for join cardinality estimation. In L. Liu, A. Reuter, K. Whang, and J. Zhang, editors, *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 20. IEEE Computer Society, 2006.

[48] U. Feige. A threshold of ln n for approximating set cover. *J. ACM*, 45(4):634–652, 1998.

[49] M. Feldman, A. Norouzi-Fard, O. Svensson, and R. Zenklusen. The one-way communication complexity of submodular maximization with applications to streaming and robustness. In K. Makarychev, Y. Makarychev, M. Tulsiani, G. Kamath, and J. Chuzhoy, editors, *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 1363–1374. ACM, 2020.

[50] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985.

[51] D. Florescu and G. Fourny. Jsoniq: The history of a query language. *IEEE Internet Comput.*, 17(5):86–90, 2013.

[52] X. Fu, T. Ghaffar, J. C. Davis, and D. Lee. Edgewise: A better stream processing engine for the edge. In D. Malkhi and D. Tsafrir, editors, *Proceedings of the 2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 929–946. USENIX Association, 2019.

[53] V. Gaede and O. Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.

[54] A. Galakatos, A. Crotty, E. Zgraggen, C. Binnig, and T. Kraska. Revisiting reuse for approximate query processing. *Proc. VLDB Endow.*, 10(10):1142–1153, 2017.

[55] M. N. Garofalakis, J. Gehrke, and R. Rastogi, editors. *Data Stream Management - Processing High-Speed Data Streams.* Data-Centric Systems and Applications. Springer, 2016.

[56] B. Gedik, R. Bordawekar, and P. S. Yu. Celljoin: a parallel stream join operator for the cell processor. *VLDB J.*, 18(2):501–519, 2009.

[57] M. Germain, K. Gregor, I. Murray, and H. Larochelle. MADE: masked autoencoder for distribution estimation. In F. R. Bach and D. M. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 881–889. JMLR.org, 2015.

[58] L. Getoor, B. Taskar, and D. Koller. Selectivity estimation using probabilistic models. In S. Mehrotra and T. K. Sellis, editors, *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 461–472. ACM, 2001.

[59] D. Gjurovski, J. Heidemann, and S. Michel. Summarizing edge-device data via core items. In S. Chiusano, T. Cerquitelli, and R. Wrembel, editors, *Advances in Databases and Information Systems - 26th European Conference, ADBIS 2022, Turin, Italy, September 5-8, 2022, Proceedings*, volume 13389 of *Lecture Notes in Computer Science*, pages 133–147. Springer, 2022.

[60] D. Gjurovski and S. Michel. Scaling out schema-free stream joins. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, pages 805–816. IEEE, 2020.

[61] D. Gjurovski and S. Michel. Natural joins for schema-free data. In Y. Chen, H. Ludwig, Y. Tu, U. M. Fayyad, X. Zhu, X. Hu, S. Byna, X. Liu, J. Zhang, S. Pan, V. Papalexakis, J. Wang, A. Cuzzocrea, and C. Ordonez, editors, *2021 IEEE International Conference on Big Data (Big Data), Orlando, FL, USA, December 15-18, 2021*, pages 2372–2380. IEEE, 2021.

[62] D. Gjurovski and S. Michel. Tuple bubbles: Learned tuple representations for tunable approximate query processing. In R. Bordawekar, O. Shmueli, Y. Amsterdamer, D. Firmani, and A. Kipf, editors, *Proceedings of the Sixth International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2023, Seattle, WA, USA, 18 June 2023*, pages 2:1–2:9. ACM, 2023.

[63] R. Gomes and A. Krause. Budgeted nonparametric learning from data streams. In J. Fürnkranz and T. Joachims, editors, *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, pages 391–398. Omnipress, 2010.

[64] I. J. Goodfellow, Y. Bengio, and A. C. Courville. *Deep Learning.* Adaptive computation and machine learning. MIT Press, 2016.

[65] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation op-

erator generalizing group-by, cross-tab, and sub totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.

[66] D. Gunopulos, G. Kollios, V. J. Tsotras, and C. Domeniconi. Approximating multi-dimensional aggregate range queries over real attributes. In W. Chen, J. F. Naughton, and P. A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 463–474. ACM, 2000.

[67] D. Gunopulos, G. Kollios, V. J. Tsotras, and C. Domeniconi. Selectivity estimators for multidimensional range queries over real attributes. *VLDB J.*, 14(2):137–154, 2005.

[68] M. Halford, P. Saint-Pierre, and F. Morvan. An approach based on bayesian networks for query selectivity estimation. In G. Li, J. Yang, J. Gama, J. Natwichai, and Y. Tong, editors, *Database Systems for Advanced Applications - 24th International Conference, DASFAA 2019, Chiang Mai, Thailand, April 22-25, 2019, Proceedings, Part II*, volume 11447 of *Lecture Notes in Computer Science*, pages 3–19. Springer, 2019.

[69] M. Halford, P. Saint-Pierre, and F. Morvan. Selectivity estimation with attribute value dependencies using linked bayesian networks. *Trans. Large Scale Data Knowl. Centered Syst.*, 46:154–188, 2020.

[70] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In W. Chen, J. F. Naughton, and P. A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 1–12. ACM, 2000.

[71] Y. Han, Z. Wu, P. Wu, R. Zhu, J. Yang, L. W. Tan, K. Zeng, G. Cong, Y. Qin, A. Pfadler, Z. Qian, J. Zhou, J. Li, and B. Cui. Cardinality estimation in DBMS: A comprehensive benchmark evaluation. *Proc. VLDB Endow.*, 15(4):752–765, 2021.

[72] Z. Hanzálek, H. Härtig, M. Castro, and M. F. Kaashoek, editors. *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013.* ACM, 2013.

[73] S. Hasan, S. Thirumuruganathan, J. Augustine, N. Koudas, and G. Das. Deep learning models for selectivity estimation of multi-attribute queries. In D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 1035–1050. ACM, 2020.

[74] R. Hayek and O. Shmueli. Improved cardinality estimation by learning queries containment rates. In A. Bonifati, Y. Zhou, M. A. V. Salles, A. Böhm, D. Olteanu, G. H. L. Fletcher, A. Khan, and B. Yang, editors, *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, pages 157–168. OpenProceedings.org, 2020.

[75] B. Hendrickson and R. W. Leland. A multi-level algorithm for partitioning graphs. In S. Karin, editor, *Proceedings Supercomputing '95, San Diego, CA, USA, December 4-8, 1995*, page 28. ACM, 1995.

[76] B. Hilprecht, A. Schmidt, M. Kulessa, A. Molina, K. Kersting, and C. Binnig. Deepdb: Learn from data, not from queries! *Proc. VLDB Endow.*, 13(7):992–1005, 2020.

[77] P. Indyk, S. Mahabadi, M. Mahdian, and V. S. Mirrokni. Composable core-sets for diversity and coverage maximization. In R. Hull and M. Grohe, editors, *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*, pages 100–108. ACM, 2014.

[78] Y. E. Ioannidis. The history of histograms (abridged). In J. C. Freytag, P. C. Lockemann, S. Abiteboul, M. J. Carey, P. G. Selinger, and A. Heuer, editors, *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003*, pages 19–30. Morgan Kaufmann, 2003.

[79] T. Kaldewey, G. M. Lohman, R. Müller, and P. B. Volk. GPU join processing revisited. In S. Chen and S. Harizopoulos, editors, *Proceedings of the Eighth International Workshop on Data Management on New Hardware, DaMoN 2012, Scottsdale, AZ, USA, May 21, 2012*, pages 55–62. ACM, 2012.

[80] S. Kandula, A. Shanbhag, A. Vitorovic, M. Olma, R. Grandl, S. Chaudhuri, and B. Ding. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. In F. Özcan, G. Koutrika, and S. Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 631–646. ACM, 2016.

[81] J. Kang, J. F. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In U. Dayal, K. Ramamritham, and T. M. Vijayaraman, editors, *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 341–352. IEEE Computer Society, 2003.

[82] N. R. Katsipoulakis, A. Labrinidis, and P. K. Chrysanthis. A holistic view of stream partitioning costs. *Proc. VLDB Endow.*, 10(11):1286–1297, 2017.

[83] L. Kaufman and P. J. Rousseeuw. Partitioning around medoids (program pam). *Finding groups in data: an introduction to cluster analysis*, 344:68–125, 1990.

[84] E. Kazemi, M. Mitrovic, M. Zadimoghaddam, S. Lattanzi, and A. Karbasi. Submodular streaming in all its glory: Tight approximation, minimum memory and low adaptive complexity. In K. Chaudhuri and R. Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 3311–3320. PMLR, 2019.

[85] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In S. Abiteboul, K. Böhm, C. Koch, and K. Tan, editors, *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 195–206. IEEE Computer Society, 2011.

[86] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, 1970.

[87] C. Kim, E. Sedlar, J. Chhugani, T. Kaldewey, A. D. Nguyen, A. D. Blas, V. W. Lee, N. Satish, and P. Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *Proc. VLDB Endow.*, 2(2):1378–1389, 2009.

[88] A. Kipf, T. Kipf, B. Radke, V. Leis, P. A. Boncz, and A. Kemper. Learned cardinalities: Estimating correlated joins with deep learning. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019.

[89] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of hash to data base machine and its architecture. *New Gener. Comput.*, 1(1):63–74, 1983.

[90] S. Krishnan, Z. Yang, K. Goldberg, J. M. Hellerstein, and I. Stoica. Learning to optimize join queries with deep reinforcement learning. *CoRR*, abs/1808.03196, 2018.

[91] M. Kulessa, A. Molina, C. Binnig, B. Hilprecht, and K. Kersting. Model-based approximate query processing. *CoRR*, abs/1811.06224, 2018.

[92] H. Lan, Z. Bao, and Y. Peng. A survey on advancing the DBMS query optimizer: Cardinality estimation, cost model, and plan enumeration. *Data Sci. Eng.*, 6(1):86–101, 2021.

[93] H. Lang, V. Leis, M. Albutiu, T. Neumann, and A. Kemper. Massively parallel numa-aware hash joins. In A. Jagatheesan, J. J. Levandoski, and T. Neumann, editors, *Proceedings of the 1st International Workshop on In Memory Data Management and Analytics, IMDM 2013, Riva Del Garda, Italy, August 26, 2013*, pages 1–12, 2013.

[94] N. D. Lawrence, M. W. Seeger, and R. Herbrich. Fast sparse gaussian process methods: The informative vector machine. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems 15 [Neural Information Processing Systems, NIPS 2002, December 9-14, 2002, Vancouver, British Columbia, Canada]*, pages 609–616. MIT Press, 2002.

[95] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, 2015.

[96] V. Leis, B. Radke, A. Gubichev, A. Kemper, and T. Neumann. Cardinality estimation done right: Index-based join sampling. In *8th Biennial Confer-*

ence on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings. www.cidrdb.org, 2017.

[97] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J. M. VanBriesen, and N. S. Glance. Cost-effective outbreak detection in networks. In P. Berkhin, R. Caruana, and X. Wu, editors, *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Jose, California, USA, August 12-15, 2007*, pages 420–429. ACM, 2007.

[98] F. Li, B. Wu, K. Yi, and Z. Zhao. Wander join: Online aggregation via random walks. In F. Özcan, G. Koutrika, and S. Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 615–629. ACM, 2016.

[99] F. Li, B. Wu, K. Yi, and Z. Zhao. Wander join and XDB: online aggregation via random walks. *ACM Trans. Database Syst.*, 44(1):2:1–2:41, 2019.

[100] G. Li, Z. Li, S. Idreos, and D. Srivastava, editors. *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 2021.

[101] X. Liang, S. Sintos, Z. Shang, and S. Krishnan. Combining aggregation and sampling (nearly) optimally for approximate query processing. In Li et al. [100], pages 1129–1141.

[102] H. Lin and J. A. Bilmes. Multi-document summarization via budgeted maximization of submodular functions. In *Human Language Technologies: Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings, June 2-4, 2010, Los Angeles, California, USA*, pages 912–920. The Association for Computational Linguistics, 2010.

[103] H. Lin and J. A. Bilmes. A class of submodular functions for document summarization. In D. Lin, Y. Matsumoto, and R. Mihalcea, editors, *The 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, Proceedings of the Conference, 19-24 June, 2011, Portland, Oregon, USA*, pages 510–520. The Association for Computer Linguistics, 2011.

[104] Q. Lin, B. C. Ooi, Z. Wang, and C. Yu. Scalable distributed stream join processing. In T. K. Sellis, S. B. Davidson, and Z. G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 811–825. ACM, 2015.

[105] C. Liu and H. Chen. A hash partition strategy for distributed query processing. In P. M. G. Apers, M. Bouzeghoub, and G. Gardarin, editors, *Advances in Database Technology - EDBT'96, 5th International Conference on Extending Database Technology, Avignon, France, March 25-29, 1996, Proceedings*, volume 1057 of *Lecture Notes in Computer Science*, pages 373–387. Springer, 1996.

[106] H. Liu, M. Xu, Z. Yu, V. Corvinelli, and C. Zuzarte. Cardinality estimation using neural networks. In J. Gould, M. Litoiu, and H. Lutfiyya, editors, *Proceedings of 25th Annual International Conference on Computer Science and Software Engineering, CASCON 2015, Markham, Ontario, Canada, 2-4 November, 2015*, pages 53–59. IBM / ACM, 2015.

[107] J. Liu, W. Dong, D. Li, and Q. Zhou. Fauce: Fast and accurate deep ensembles with uncertainty for cardinality estimation. *Proc. VLDB Endow.*, 14(11):1950–1963, 2021.

[108] M. Luber, L. Spinello, and K. O. Arras. People tracking in RGB-D data with on-line boosted target models. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2011, San Francisco, CA, USA, September 25-30, 2011*, pages 3844–3849. IEEE, 2011.

[109] Q. Ma and P. Triantafillou. Dbest: Revisiting approximate query processing engines with machine learning models. In P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1553–1570. ACM, 2019.

[110] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing main-memory join on modern hardware. *IEEE Trans. Knowl. Data Eng.*, 14(4):709–730, 2002.

[111] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: A learned query optimizer. *Proc. VLDB Endow.*, 12(11):1705–1718, 2019.

[112] R. Marcus and O. Papaemmanouil. Deep reinforcement learning for join order enumeration. In R. Bordawekar and O. Shmueli, editors, *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2018, Houston, TX, USA, June 10, 2018*, pages 3:1–3:4. ACM, 2018.

[113] R. Marcus and O. Papaemmanouil. Towards a hands-free query optimizer through deep learning. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019.

[114] Z. Meng, P. Wu, G. Cong, R. Zhu, and S. Ma. Unsupervised selectivity estimation by integrating gaussian mixture models and an autoregressive model. In J. Stoyanovich, J. Teubner, P. Guagliardo, M. Nikolic, A. Pieris, J. Mühlig, F. Özcan, S. Schelter, H. V. Jagadish, and M. Zhang, editors, *Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022*, pages 2:247–2:259. OpenProceedings.org, 2022.

[115] V. S. Mirrokni and M. Zadimoghaddam. Randomized composable coresets for distributed submodular maximization. In R. A. Servedio and R. Rubinfeld, editors, *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 153–162. ACM, 2015.

[116] B. Mirzasoleiman, A. Karbasi, R. Sarkar, and A. Krause. Distributed submodular maximization: Identifying representative elements in massive data. In C. J. C. Burges, L. Bottou, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, pages 2049–2057, 2013.

[117] M. Muralikrishna and D. J. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In H. Boral and P. Larson, editors, *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 1-3, 1988*, pages 28–36. ACM Press, 1988.

[118] K. P. Murphy. *Probabilistic Machine Learning: Advanced Topics*. MIT Press, 2023.

[119] MySQL Team. Cardinality estimation (mysql). `https://dev.mysql.com/doc/refman/8.4/en/optimizer-statistics.html`, May 2024. Accessed 17 October 2024.

[120] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska. Learning multi-dimensional indexes. In D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 985–1000. ACM, 2020.

[121] P. Negi, R. Marcus, A. Kipf, H. Mao, N. Tatbul, T. Kraska, and M. Alizadeh. Flow-loss: Learning cardinality estimates that matter. *Proc. VLDB Endow.*, 14(11):2019–2032, 2021.

[122] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher. An analysis of approximations for maximizing submodular set functions - I. *Math. Program.*, 14(1):265–294, 1978.

[123] T. Neumann and S. Michel. Smooth interpolating histograms with error guarantees. In W. A. Gray, K. G. Jeffery, and J. Shao, editors, *Sharing Data, Information and Knowledge, 25th British National Conference on Databases, BNCOD 25, Cardiff, UK, July 7-10, 2008. Proceedings*, volume 5071 of *Lecture Notes in Computer Science*, pages 126–138. Springer, 2008.

[124] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In Sellis et al. [146], pages 949–960.

[125] K. W. Ong, Y. Papakonstantinou, and R. Vernoux. The SQL++ semi-structured data model and query language: A capabilities survey of sql-on-hadoop, nosql and newsql databases. *CoRR*, abs/1405.3631, 2014.

[126] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. Learning state representations for query optimization with deep reinforcement learning. In S. Schelter, S. Seufert, and A. Kumar, editors, *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning, DEEM@SIGMOD 2018, Houston, TX, USA, June 15, 2018*, pages 4:1–4:4. ACM, 2018.

[127] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. An empirical analysis of deep learning for cardinality estimation. *CoRR*, abs/1905.06425, 2019.

[128] Y. Park, B. Mozafari, J. Sorenson, and J. Wang. Verdictdb: Universalizing approximate query processing. In Das et al. [36], pages 1461–1476.

[129] J. Pearl. *Probabilistic reasoning in intelligent systems - networks of plausible inference*. Morgan Kaufmann series in representation and reasoning. Morgan Kaufmann, 1989.

[130] J. Peng, D. Zhang, J. Wang, and J. Pei. AQP++: connecting approximate query processing with aggregate precomputation for interactive analytics. In Das et al. [36], pages 1477–1492.

[131] T. Pitoura and P. Triantafillou. Load distribution fairness in P2P data management systems. In R. Chirkova, A. Dogac, M. T. Özsu, and T. K. Sellis, editors, *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 396–405. IEEE Computer Society, 2007.

[132] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 486–495. Morgan Kaufmann, 1997.

[133] PostgreSQL Global Development Group. 76.1. row estimation examples. `https://www.postgresql.org/docs/current/row-estimation-examples.html`, Feb 2024. Accessed 17 October 2024.

[134] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang. Timestream: reliable stream computation in the cloud. In Hanzálek et al. [72], pages 1–14.

[135] D. L. Quoc, I. E. Akkus, P. Bhatotia, S. Blanas, R. Chen, C. Fetzer, and T. Strufe. Approxjoin: Approximate distributed joins. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*, pages 426–438. ACM, 2018.

[136] M. Raasveldt and H. Mühleisen. Duckdb: an embeddable analytical database. In P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1981–1984. ACM, 2019.

[137] C. E. Rasmussen and C. K. I. Williams. *Gaussian processes for machine learning*. Adaptive computation and machine learning. MIT Press, 2006.

[138] M. Reif and T. Neumann. A scalable and generic approach to range joins. *Proc. VLDB Endow.*, 15(11):3018–3030, 2022.

[139] P. Roy, J. Teubner, and R. Gemulla. Low-latency handshake join. *Proc. VLDB Endow.*, 7(9):709–720, 2014.

[140] H. Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[141] P. Sanders and C. Schulz. Engineering multilevel graph partitioning algorithms. In C. Demetrescu and M. M. Halldórsson, editors, *Algorithms - ESA 2011 - 19th Annual European Symposium, Saarbrücken, Germany, September 5-9, 2011. Proceedings*, volume 6942 of *Lecture Notes in Computer Science*, pages 469–480. Springer, 2011.

[142] N. Schäfer and S. Michel. JODA: A vertically scalable, lightweight JSON processor for big data transformations. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, pages 1726–1729. IEEE, 2020.

[143] N. Schäfer and S. Michel. Partially materializable delta trees for efficient data wrangling of semi-structured contents. In A. Bonifati, Y. Zhou, M. A. V. Salles, A. Böhm, D. Olteanu, G. H. L. Fletcher, A. Khan, and B. Yang, editors, *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, pages 399–402. OpenProceedings.org, 2020.

[144] I. J. Schoenberg. Metric spaces and completely monotone functions. *Annals of Mathematics*, 39(4):811–841, 1938.

[145] M. Seeger. Greedy forward selection in the informative vector machine. *Technical report, UC Berkeley*, 2004.

[146] T. K. Sellis, R. J. Miller, A. Kementsietsidis, and Y. Velegrakis, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011.* ACM, 2011.

[147] L. Spinello and K. O. Arras. People detection in RGB-D data. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2011, San Francisco, CA, USA, September 25-30, 2011*, pages 3838–3843. IEEE, 2011.

[148] G. A. Stafford. Environmental sensor telemetry data. `https://www.kaggle.com/datasets/garystafford/environmental-sensor-data-132k`, 2020. Accessed: 2022-04-28.

[149] J. W. Stamos and H. C. Young. A symmetric fragment and replicate algorithm for distributed joins. *IEEE Trans. Parallel Distributed Syst.*, 4(12):1345–1354, 1993.

[150] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - db2's learning optimizer. In P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, editors, *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 19–28. Morgan Kaufmann, 2001.

[151] D. Tahara, T. Diamond, and D. J. Abadi. Sinew: a SQL system for multi-structured data. In C. E. Dyreson, F. Li, and M. T. Özsu, edi-

tors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 815–826. ACM, 2014.

[152] P. Tan, M. S. Steinbach, A. Karpatne, and V. Kumar. *Introduction to Data Mining (Second Edition)*. Pearson, 2019.

[153] J. Teubner and R. Müller. How soccer players would do stream joins. In Sellis et al. [146], pages 625–636.

[154] S. Thirumuruganathan, S. Hasan, N. Koudas, and G. Das. Approximate query processing for data exploration using deep generative models. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, pages 1309–1320. IEEE, 2020.

[155] K. Tzoumas, A. Deshpande, and C. S. Jensen. Lightweight graphical models for selectivity estimation without independence assumptions. *Proc. VLDB Endow.*, 4(11):852–863, 2011.

[156] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[157] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.

[158] B. Walenz, S. Sintos, S. Roy, and J. Yang. Learning to sample: Counting with complex queries. *Proc. VLDB Endow.*, 13(3):390–402, 2019.

[159] S. Wang and E. A. Rundensteiner. Scalable stream join processing with expensive predicates: workload distribution and adaptation by time-slicing. In M. L. Kersten, B. Novikov, J. Teubner, V. Polutin, and S. Manegold, editors, *EDBT 2009, 12th International Conference on Extending Database Technology, Saint Petersburg, Russia, March 24-26, 2009, Proceedings*, volume 360 of *ACM International Conference Proceeding Series*, pages 299–310. ACM, 2009.

[160] T. Wang and C. Chan. Improved correlated sampling for join size estimation. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, pages 325–336. IEEE, 2020.

[161] WilliamDAssafMSFT et al. Cardinality estimation (sql server). `https://learn.microsoft.com/en-us/sql/relational-databases/performance/cardinality-estimation-sql-server?view=sql-server-ver16`, Aug 2024. Accessed 17 October 2024.

[162] L. Woltmann, C. Hartmann, M. Thiele, D. Habich, and W. Lehner. Cardinality estimation with local deep learning models. In R. Bordawekar and O. Shmueli, editors, *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2019, Amsterdam, The Netherlands, July 5, 2019*, pages 5:1–5:8. ACM, 2019.

[163] S. Wu, B. C. Ooi, and K. Tan. Continuous sampling for online aggregation over multiple queries. In A. K. Elmagarmid and D. Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 651–662. ACM, 2010.

[164] Z. Wu, P. Negi, M. Alizadeh, T. Kraska, and S. Madden. Factorjoin: A new cardinality estimation framework for join queries. *Proc. ACM Manag. Data*, 1(1):41:1–41:27, 2023.

[165] Z. Wu and A. Shaikhha. Bayescard: A unified bayesian framework for cardinality estimation. *CoRR*, abs/2012.14743, 2020.

[166] Z. Yang, A. Kamsetty, S. Luan, E. Liang, Y. Duan, X. Chen, and I. Stoica. Neurocard: One cardinality estimator for all tables. *Proc. VLDB Endow.*, 14(1):61–73, 2020.

[167] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, X. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica. Deep unsupervised cardinality estimation. *Proc. VLDB Endow.*, 13(3):279–292, 2019.

[168] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: fault-tolerant streaming computation at scale. In M. Kaminsky and M. Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 423–438. ACM, 2013.

[169] E. Zgraggen, A. Galakatos, A. Crotty, J. Fekete, and T. Kraska. How progressive visualizations affect exploratory analysis. *IEEE Trans. Vis. Comput. Graph.*, 23(8):1977–1987, 2017.

[170] M. Zhang and H. Wang. LAQP: learning-based approximate query processing. *Inf. Sci.*, 546:1113–1134, 2021.

[171] N. L. Zhang and D. Poole. A simple approach to bayesian network computations. In *Proc. of the Tenth Canadian Conference on Artificial Intelligence*, 1994.

[172] J. Zhao, P. Wang, J. Tao, S. Zhang, and J. C. S. Lui. Continuously tracking core items in data streams with probabilistic decays. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, pages 769–780. IEEE, 2020.

[173] Z. Zhao, R. Christensen, F. Li, X. Hu, and K. Yi. Random sampling over joins revisited. In Das et al. [36], pages 1525–1539.

[174] Y. Zhou, Y. Yan, F. Yu, and A. Zhou. Pmjoin: Optimizing distributed multi-way stream joins by stream partitioning. In M. Lee, K. Tan, and V. Wuwongse, editors, *Database Systems for Advanced Applications, 11th International Conference, DASFAA 2006, Singapore, April 12-15, 2006, Proceedings*, volume 3882 of *Lecture Notes in Computer Science*, pages 325–341. Springer, 2006.

[175] R. Zhu, Z. Wu, Y. Han, K. Zeng, A. Pfadler, Z. Qian, J. Zhou, and B. Cui. FLAT: fast, lightweight and accurate method for cardinality estimation. *Proc. VLDB Endow.*, 14(9):1489–1502, 2021.

[176] H. Zhuang, R. Rahman, X. Hu, T. Guo, P. Hui, and K. Aberer. Data summarization with social contexts. In S. Mukhopadhyay, C. Zhai, E. Bertino, F. Crestani, J. Mostafa, J. Tang, L. Si, X. Zhou, Y. Chang, Y. Li, and P. Sondhi, editors, *Proceedings of the 25th ACM International Conference on Information and Knowledge Management, CIKM 2016, Indianapolis, IN, USA, October 24-28, 2016*, pages 397–406. ACM, 2016.

Damjan Gjurovski

Master of Science in Computer Science

Education

07/2019 - Today **Doctoral Studies**, RPTU Kaiserslautern-Landau
Thesis Title: Query Processing and Cardinality Estimation over Data Summaries

10/2016 - 03/2019 **MSc. Computer Science**, TU Kaiserslautern
Thesis Title: Scaling out Equi Joins over Massive Semi-Structured Data Streams
Major: Information systems

09/2012 - 07/2016 **BSc. Computer Science**, Faculty of Computer Science and Engineering, Skopje (FINKI)
Thesis Title: Development of a Global Open Drug Data Registry
Major: Computer Science and Engineering

Work Experience

01/2020 - Today **Research Associate**, TU Kaiserslautern / RPTU Kaiserslautern-Landau
Employment within the framework of the doctorate.
Responsibilities: Organization of exercises, seminars, projects, and, supervision of theses. Work within the DigiVine project.

05/2017 - 04/2019 **Student Assistant**, Fraunhofer IESE, Kaiserslautern
Responsibilities: Various projects, including, software architecture and machine learning for automobile automation, mining association rules from product configurations, and others.

06/2016 - 10/2016 **Front-end Developer**, Bitcoinaverage, Remote
Responsibilities: Design and develop front-end for web app UI, implement SEO techniques, introduce new plots for representing data.

damjan.gjurovski@cs.rptu.de