SplitDF: Splitting Dataframes for Memory-Efficient Data Analysis

Aarati Kakaraparthy
University of Wisconsin, Madison
aaratik@cs.wisc.edu

Jignesh M. Patel
Carnegie Mellon University
jignesh@cmu.edu

ABSTRACT

Dataframe is a popular construct in data analysis libraries that offers a tabular view of the data. However, data within a dataframe often has redundancy, which can lead to high memory utilization of data analysis libraries. Inspired by the process of normalization in relational database systems, we propose a technique called *splitting* that can be applied to tabular data to reduce redundancy. Splitting involves performing lossless join decomposition by explicitly adding joining keys, and unlike normalization, splitting can be applied to tabular data without the need to perform functional dependency discovery. A *split* dataframe provides the same unified tabular view to the data, while internally operating on split data to improve memory efficiency. We develop SplitDF, an implementation of split dataframes in Ibis for DuckDB backend, which enables data analysis on split data with minimal changes to the Ibis API. Generation of split tabular data is automated using an algorithm `SplitGen` implemented in Velox. In our analysis involving ten handwritten notebooks running on SplitDF, we observe a reduction in memory usage of 19-61% when operating on split data as compared to operating on original data.

PVLDB Reference Format:
Aarati Kakaraparthy and Jignesh M. Patel. SplitDF: Splitting Dataframes for Memory-Efficient Data Analysis. PVLDB, 17(9): 2175 - 2184, 2024.
doi:10.14778/3665844.3665849

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/UWQuickstep/splitting/.

1 INTRODUCTION

Data scientists often deal with large datasets that are tabular, distributed in open formats such as CSV, JSON, and Parquet [22]. The working environment for many data scientists are Python notebooks, either hosted on a laptop or virtualized cloud containers. Data scientists perform tasks such as data cleaning, feature engineering, and exploration using data analysis libraries such as Pandas [26], Ibis [20], Koalas [23], etc. to name a few.

Dataframe is a key tabular construct provided by data analysis libraries. While different libraries offer different abstractions for the dataframe object [55], one common aspect of all dataframe libraries is that they operate on a tabular view of the data loaded directly from raw data files. Dataframes have no semantics of normalization associated with them. While loading into dataframes directly from raw data files is convenient for data analysis, dataframe libraries are known to suffer from high memory utilization [13, 30, 60] as we also see in our work (Fig. 1).

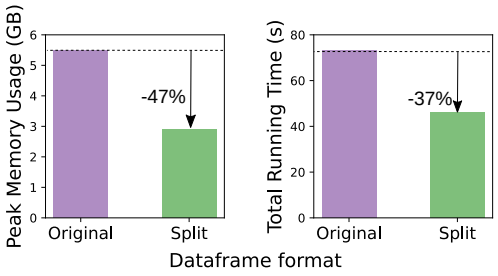


Figure 1: Peak memory usage and running time of a notebook implemented on the NYC parking tickets 2014 CSV dataset of size 1.9 GB. When operating on the original dataframe, the peak memory usage of the notebook is 3× the size of the raw CSV data. Operating on split dataframe reduces the memory usage by 47% and the running time by 37%.

A key contributor to high memory usage of data analysis libraries is redundancy in the data. Redundancy arises when the data has correlated/dependent attributes, or attributes with few unique values. Relational database systems have employed normalization to systematically identify and reduce redundancy in the data by capturing functional dependencies between attributes. Designing an effective relational schema involves discovering functional dependencies in the data and taking steps to conform to a desired normal form to reduce the redundancy and improve the integrity of the data, and these steps often require the involvement of a database administrator. The key question we consider in this paper is: *Can specific principles of normalization from database theory be applied to dataframes to improve storage efficiency and data analysis speed, with minimal effort on the part of the data scientist?*

We propose a technique called *splitting* which is inspired by the lossless-join decomposition mechanism [57] from database normalization theory. Splitting can be automatically performed on tabular data, and does not require functional dependency discovery (FD) and schema design on part of the user. A split dataframe internally operates on split data, while exposing the same unified tabular interface as if operating on the original data. To demonstrate the effectiveness of splitting for improving memory efficiency of dataframe libraries, we make the following contributions in the paper:

- *Formal definition of splitting*: In a nutshell, splitting involves performing a lossless-join decomposition [24] on a table by explicitly introducing joining keys (Fig. 2). Splitting can be performed on arbitrary groups of attributes without requiring FD discovery.
- *Generating automatic splits with* `SplitGen`: We developed an algorithm `SplitGen` that generates attribute groups for splitting using statistics from the data, and does not require the user to

Proceedings of the VLDB Endowment, Vol. 17, No. 9 ISSN 2150-8097.
doi:10.14778/3665844.3665849

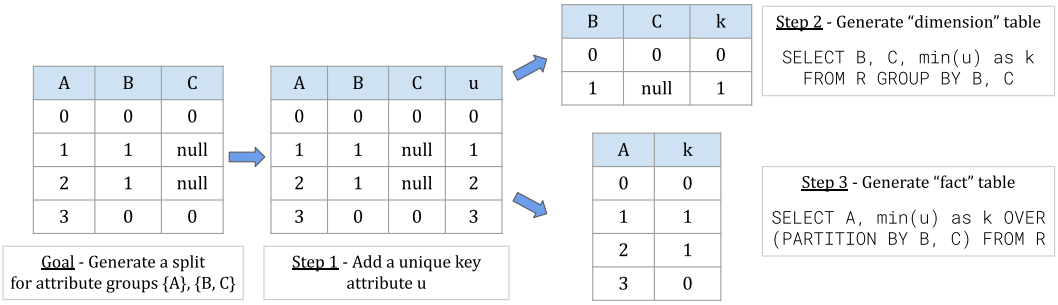


Figure 2: Two-way splitting of a table R = {A, B, C} into tables with schema {B, C, k} and {A, k}, where k is the joining key. In this example, the generated split satisfies the functional dependency k → {B, C}. *Splitting* is an efficient form of automatic lossless join decomposition, i.e., the original table can be recovered by joining the split tables on attribute k. Generating the split tables requires using aggregation and window operations. N-way splits can be generated by splitting a table (N-1) times.

perform schema design. We implemented `SplitGen` in Velox [54] to automatically generate split CSV files.

- *Splitting dataframes in Ibis with SplitDF*: SplitDF is an implementation of split dataframes in Ibis [20] for DuckDB [56] backend. SplitDF makes minimal changes to the Ibis API, while improving memory efficiency by operating on split data under the hood.
- *Evaluation on open datasets*: We conduct our evaluation on top-voted CSV datasets on Kaggle [22]. We implemented ten notebooks for three datasets with sizes ranging from 1.2GB to 4.8GB. Running the notebooks on SplitDF produces a reduction in memory usage of 19-61% and a reduction in running time of 1-58% when operating on split data as compared to operating on original data. The improvement in memory efficiency stems from the effectiveness of splitting. Running `SplitGen` on twelve open CSV datasets shows that for six out of the twelve datasets tested, we obtain more than an 40% reduction in total size from splitting.

We give a formal specification of splitting in §2, followed by describing `SplitDF` in §3. We describe the `SplitGen` algorithm in §4, followed by evaluation in §5. We discuss related work in §6 and our conclusions are in §7. Through our work, we aim to demonstrate the benefits of the proposed splitting mechanism in improving the memory efficiency of dataframe libraries.

2 WHAT IS SPLITTING?

In this section, we formally define splitting in §2 followed by describing how to generate a split in a relational engine in §2.2. We discuss the differences between splitting and normalization in §2.3, and describe our vision splitting dataframes in §2.4.

2.1 Definition

Given a relation schema R, a two-way split of R into schemas with attribute sets X $\bigcup\{k\}$ and Y $\bigcup\{k\}$ is such that

- X $\bigcup$ Y = R, X $\bigcap$ Y = $\emptyset$
- $k \notin$ R
- either of the FD $k \rightarrow$ X or $k \rightarrow$ Y hold □

In other words, a two-way split is a lossless decomposition of the schema R $\bigcup\{k\}$, where the property of losslessness follows from the constraint that either $k \rightarrow$ X or $k \rightarrow$ Y hold [57]. The unique aspect of splitting is that we explicitly introduce a "joining" key k in the schema that satisfies either $k \rightarrow$ X or $k \rightarrow$ Y allowing one to perform the split for any disjoint attribute groups (X, Y). Given an instance r of relation R, and a two-way split of r into x' and y' which are respectively instances of schemas X $\bigcup\{k\}$ and Y $\bigcup\{k\}$ that satisfy the above mentioned criterion, r can be recovered as $\pi_R(x' \bowtie y') = r$. A two-way split can easily be generalized to obtain an n-way split of a relation schema R.

2.2 Generating a Split

Fig. 2 shows the steps involved in generating a two-way split of a relation R = {A, B, C} into attribute groups {A} and {B, C}. The first step is to add an unique key attribute (u) to the relation which is supported by most major database engines [1, 11]. To generate a split satisfying the FD k → {B, C}, the dimension table with schema {B, C, k} is generated by performing an aggregation over attributes {B, C} of R. Note that k is the primary key of the dimension table. The fact table is generated using window operation on the relation schema R $\bigcup$ {u} over attributes {B, C} . Note that both these operations involve simple aggregations. To generate an n-way split, one could recursively apply splitting to the fact table to generate *(n-1)* dimension tables. The approach shown in Fig. 2 can be implemented in any relational DBMS.

2.3 Splitting vs Normalization

Both splitting and normalization involve decomposition of a relation to reduce redundancy in the schema. However, normalization also accounts for integrity constraints of the database to guard against update, insertion and deletion anomalies. Mining functional dependencies (FDs) [52] is an important step in normalization, and these FDs are used for generating the database schema in normal forms [37, 38]. Thus, two properties of decompositions that are of interest in the context of normalization are lossless-join and dependency-preservation [57].

However, guarding against insertion, update, and delete anomalies is not a primary goal of data analysis, which often involves operations such as data exploration, cleaning, and handling null values. Thus, the key property of decompositions that is of interest in the case of data analysis is lossless-join. Splitting enables exploration of attribute groups that can reduce the overall redundancy, thus improving memory-efficiency of data analysis pipelines. Unlike normalization, splitting can be performed without functional dependency discovery and schema design on part of the user.

2.4 Splitting Dataframes

Dataframes are popular tabular structures used in data analysis libraries. Unlike a database administrator, data scientists do not perform schema design, which would require performing functional dependency discovery and normalization. Data is directly loaded from raw tabular data files into a dataframe. Thus, we propose splitting dataframes "under the hood", i.e., exposing the same unified tabular interface to the data scientist as if loaded directly from the raw data file, while internally the dataframe operates on split data.

Two major benefits of splitting are – (1) it reduces redundancy from the data, and (2) it can be applied to raw data files using automated methods (see §4) without requiring schema design. Typically, tabular data is distributed in open formats such as CSV, Parquet, and JSON [22], and major relational database engines support loading and storing data from these formats [21]. *Split* data files can be generated using a relational engine, where a split file is a collection of (ideally) smaller files corresponding to the fact and dimension tables generated during splitting. Thus, splitting can be performed on raw tabular data without requiring manual intervention from the data scientist. The data scientist can operate on the split data by loading it into a split dataframe that exposes a unified tabular representation to the user, as if operating on the original data file.

3 SPLITTING DATAFRAMES IN IBIS

In this section, we describe salient features of the Ibis library (§3.1), followed by describing SplitDF, our implementation of split dataframes in Ibis for the DuckDB backend (§3.2).

3.1 The Ibis library

The Ibis [20] library enables working on data from over fifteen backend engines in a dataframe environment, with DuckDB being the default backend. Ibis allows working on large datasets stored in relational databases or big data systems. This is an increasingly popular trend in data science libraries, also adopted by other systems [43, 46, 58] to enable working on larger than memory datasets. Thus, we chose Ibis as the system of evaluation given its growing popularity as a unified dataframe interface for data analysis.

3.2 SplitDF

We develop SplitDF, an implementation of split dataframes in Ibis for DuckDB backend. The approach taken by SplitDF can be applied to Ibis for other relational backend engines as well. SplitDF has the following key features:

- *Exposing split data as a unified view*: To retain the same unified tabular dataframe interface for the user, we load the split files in the backend engine and declare a view. Ibis does not differentiate between tables and views, so the users can operate on a dataframe corresponding to the unified view agnostic to the underlying storage format.
- *The query rewriting layer*: We implemented a query rewriting layer in Ibis which transparently generates optimized SQL queries when operating on split data. The query rewriting layer maintains information about the underlying schema of the data, and generates SQL queries such that only the required dimension tables that contain attributes referenced in the query are joined with the fact table when operating on split data. While one might expect the query optimizer in the backend engine to perform said optimization, our analysis showed that the optimization is missing in prominent database engines, namely PostgreSQL and DuckDB. The query rewriting layer internally uses the SQLGlot transpiler library [29] to generate optimized SQL queries.
- *Minimal changes to the user experience*: The only API change made by SplitDF is registering the schema of the data with Ibis to generate efficient SQL queries using the query rewriting layer. The user can conduct data analysis on a single dataframe corresponding to the unified tabular view of the data, when operating on split data under the hood.

```
import ibis
def init_from_csv(dbname, tablename, csv_file):
  '''Load from csv_file into table tablename'''
  ...
  return schema

def init_from_split_csv(
  dbname, tablename, split_csv_folder):
  '''Load the split files as individual tables,
  and declare a view with name tablename'''
  ...
  return schema

### 1. Initializing DuckDB backend ###
dbname = "us_accidents.db"
tablename = "accidents"
### Default - Load CSV file into backend ###
# csv_file = "US_Accidents_Dec21_updated.csv"
# schema = init_from_csv(
#   dbname, tablename, csv_file)

### Split - Load split CSV into backend ###
split_folder = "US_Accidents_Dec21_updated_split/"
schema = init_from_split_csv(
  dbname, tablename, split_folder)

### 2. Register the schema with Ibis ###
con = ibis.duckdb.connect(dbname)
con.register_schema(schema)

### 3. Proceed with Data Analysis Agnostic ###
###    to the underlying format            ###
df = con.table(tablename)
agg = df.group_by('State').aggregate(df.count())
```

Listing 1: Example of data analysis with SplitDF, our implementation of split dataframes in Ibis with DuckDB backend. After initializing the backend (step 1) and registering the schema with Ibis (step 2), data analysis proceeds agnostic to the underlying storage format. The only API change SplitDF makes is registering the schema (step 2). SplitDF allows operation on both original and split CSV data with no changes to the data scientist's experience.

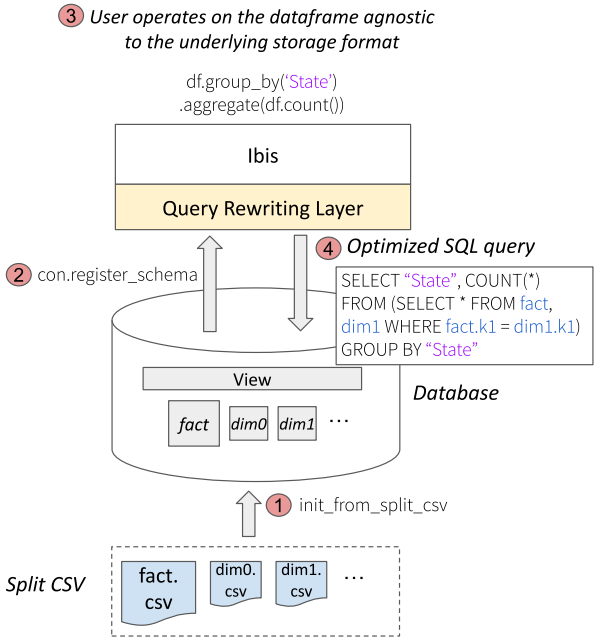


Figure 3: Architecture of SplitDF when operating on split data. Split CSV data is loaded into the backend database and exposed as a view to Ibis. The schema of the view is registered with the Ibis query rewriting layer (introduced in our work), which generates efficient SQL queries depending on the schema of the data (split vs unified). The user can conduct data analysis agnostic to the underlying storage format.

Listing 3 shows the programmer's experience when working with SplitDF. After (1) setting up the backend, and (2) registering the schema with Ibis, (3) data analysis can proceed agnostic to the underlying layout of data. Thus, SplitDF makes minimal changes to the Ibis API. The query rewriting layer generates efficient SQL for split data by joining only the required dimension tables with the fact table as shown in Fig. 3.

4 GENERATING AUTOMATIC SPLITS

A critical aspect of splitting is the choice of attribute groups such that redundancy in the data is reduced. We propose a greedy algorithm to find attribute groups that reduces the total size of the split data (§4.1). We also describe an implementation of this algorithm in Velox [54] to generate split CSV files (§4.2).

4.1 `SplitGen`: A greedy algorithm

The goal of generating splits is to reduce the redundancy in the data. We propose a greedy algorithm `SplitGen` which can be automatically executed on a table without requiring functional dependency discovery and schema design on part of the user. Instead, `SplitGen` utilizes statistics about the data to produce attribute groups for splitting. Splitting can subsequently be performed in a relational engine as shown in Fig. 2. Algorithm 1 describes the `SplitGen` algorithm, which generates attribute groups for the dimension tables (*dims*) and the fact table ($fact$) for splitting. The algorithm has the following components/steps:

Algorithm 1 `SplitGen`: Generating Attribute Groups for Splitting

```
1:  procedure GenAttributeGroups(t)                    ▷ t is a table
2:      attrs ← t.attributes
3:      nrows ← t.nrows
4:      distinct_count ← [CountDistinct(t[a]) for a in attrs]
5:                  ▷ Sorted by increasing value of distinct count
6:      attrs ← sort(attrs, key = distinct_count)
7:      distinct_count ← sort(distinct_count)
8:      max_size ← [MaxValueSize(t[a]) for a in attrs]
9:      avg_size ← [AvgValueSize(t[a] for a in attrs]
10:     stats ← (nrows, attrs, distinct_count, max_size, avg_size)
11:
12:     attr_group ← {}, dims ← [], fact ← [], i ← 0
13:     while i < len(attrs) do
14:         candidate ← attr_group.add(attrs[i])
15:         estimated_size ← EstimateSplitSize(candidate, stats)
16:         actual_size ← ActualSize(candidate, stats)
17:         if estimated_size < actual_size then
18:             attr_group ← candidate
19:             i ← i + 1              ▷ Try adding the next attribute
20:         else if size(attr_group) > 0 then
21:             dims.add(attr_group)
22:             attr_group ← {}                    ▷ Start a new group
23:         else
24:             fact.add(attrs[i])       ▷ attr[i] could not be split
25:             i ← i + 1
26:         end if
27:     end while
28:     return (dims, fact)
29: end procedure
30:
31: procedure ActualSize(candidate, stats)
32:     (nrows, attrs, distinct_count, max_size, avg_size) = stats
33:     pos ← attrs.get_indexes(candidate)
34:     size ← 0
35:     for i in pos do
36:         size ← size + nrows × avg_size[i]
37:     end for
38:     return size
39: end procedure
40:
41: procedure EstimateSplitSize(candidate, stats)
42:     (nrows, attrs, distinct_count, max_size, avg_size) = stats
43:     pos ← attrs.get_indexes(candidate)
44:     est_nrows ← 1
45:     est_tuple_size ← 0
46:     for i in pos do
47:         est_nrows ← est_nrows × distinct_count[i]
48:         est_tuple_size ← est_tuple_size + max_size[i]
49:     end for
50:     est_tuple_size ← est_tuple_size + 8  ▷ 8-byte joining key
51:     size ← est_tuple_size × est_nrows + nrows × 8
52:     return size
53: end procedure
```

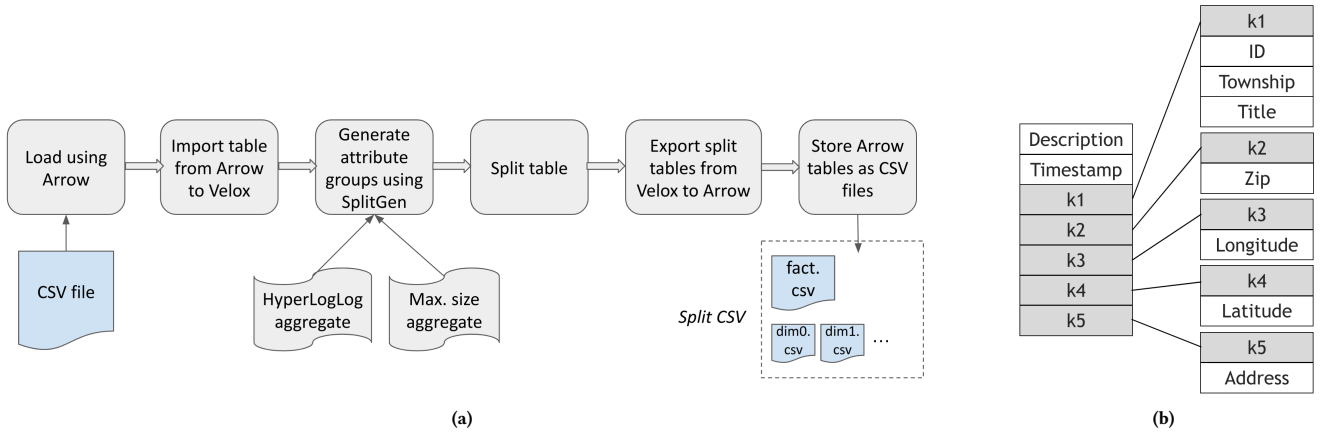


Figure 4: We developed a fully automatic module that implements `SplitGen` (cf. §4.1) using the HyperLogLog approximate count distinct aggregate function [42] in Velox. The workflow of the module is shown in (a). The split schema generated for the 911 dataset (see Table 1) is shown in (b), which yields 33% reduction in raw CSV file size compared to the original dataset.

(1) *Statistics*: The `SplitGen` algorithm utilizes three key statistics about the data: the number of distinct values, the maximum value size, and the average value size of each of the attributes.
(2) *Sliding window over attributes*: The attributes are sorted in ascending order of their distinct count. The algorithm attempts to group together attributes starting from the attribute with least number of distinct values. For each candidate attribute group, the size of the split is estimated and compared to the actual size of the attribute group. The algorithm continues to add attributes to the candidate attribute group, until the estimated split size is less than the actual size.
(3) *Generating attribute groups for dimension and fact tables*: Attribute groups for which the estimated split size is less than the actual size are added to the *dims* array, which will correspond to a dimension table. Note that dictionary encoding is a special case of this algorithm when the size of the attribute group is one. Any attributes that are not estimated to generate benefit from dictionary encoding (which is the minimal split possible) are retained in the fact table.
(4) *Estimating the size of the split*: The algorithm uses a conservative estimate of the size of the generated dimension table. The cardinality of the dimension table is estimated as the product of number distinct values of each of the attributes (which is the upper limit as not all combinations of attribute values might occur in the data) times the tuple size (estimated as the sum of the maximum value size for each of the attributes, plus the size of the joining key, which is again an upper limit). Extra space needed for the joining key attribute in the fact table is also accounted for. Thus, the estimated size of the split is an upper limit on the real size of the split.

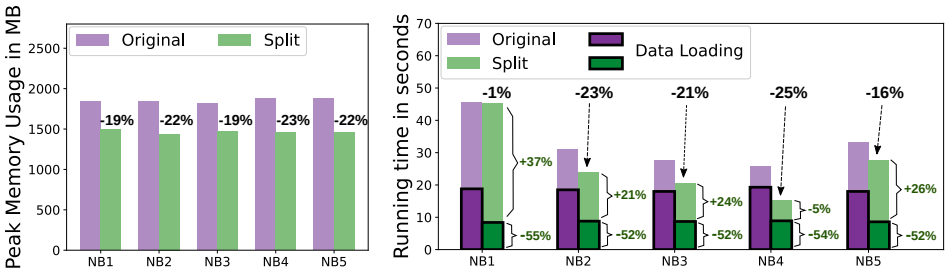
The algorithm is guaranteed to generate attribute groups for splitting that lead to a net reduction in size, as the estimated size of the split assumes independence between attributes to estimate the cardinality of the split table, and uses the max value size of each attribute to estimate the tuple size, both of which are conservative estimates. The statistics utilized by `SplitGen` (distinct values and max value size) can be obtained by performing a single pass over the data, i.e., using $O(N)$ time and space where N is the number of rows. The next step involves sorting the attributes by their distinct value count, and the complexity of this part of the algorithm is $O(a \cdot log(a))$, where a is the number of attributes. While running a sliding window to generate attributes groups, each attribute is considered at most twice to be added to a candidate attribute group, and the complexity of this part of the algorithm is $O(a)$. Thus, the overall complexity of the algorithm is $O(N + a \cdot log(a))$.

4.2 Splitting CSV files in Velox

We developed a module in Velox [54] that automatically generates split CSV files. A split CSV file is a collection of CSV files corresponding to the fact and dimension tables generated during splitting. Below are the important components of our implementation:

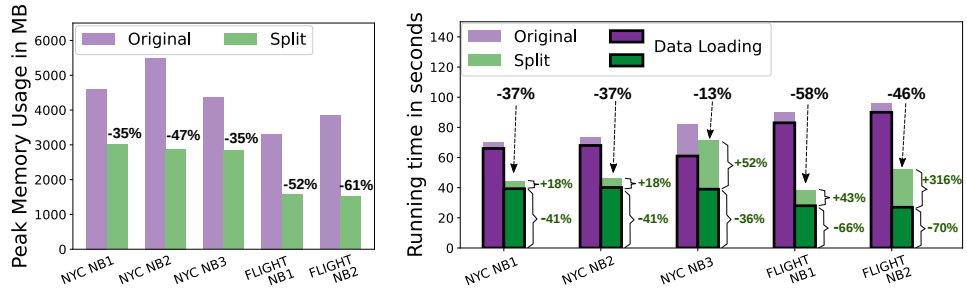
- *Reading/Writing CSV files using Apache Arrow*: Velox currently does not support reading/writing CSV files. The module use Apache Arrow [6] to ingest CSV files and write split CSV files.
- *Implementing* `SplitGen`: The statistics utlized by `SplitGen` are obtained using aggregate functions in Velox. To estimate the distinct count of each attribute, we use the HyperLogLog [41, 42] aggregate function in Velox, which provides an estimate of the distinct count with a standard error of 2.3%. Thus, our implementation does not provide the strict guarantee of yielding a split that produces a net reduction in size, but in practice we find have found it to produce effective splits (see §5.2).
- *Generating split tables*: The module uses aggregation and window operations as shown in Fig. 2 to generate dimension and fact tables respectively. To generate N-way splits, splitting is recursively applied to generated fact table (N-1) times.

Thus, splitting can be applied automatically to CSV files without manual intervention and schema design using the developed module in Velox (see Fig. 4 for the workflow of the module, and an example of a split schema). In general, splitting is performed using aggregate and window operations (see Fig. 2), and can be implemented in any relational engine that supports these operations.



Notebooks for ACCIDENT *dataset*	
Name	**Description of Operations**
NB1	Aggregations, adding new features, correlation between attributes
NB2	Aggregations, order by
NB3	Aggregations, feature engineering, dropping null values
NB4	Aggregations, dropping null values
NB5	Aggregations, adding new features, filling/dropping null values

Figure 5: Notebooks for the ACCIDENT dataset. When operating on the split dataframes, we observe a 19-23% reduction in peak memory usage across all notebooks, while the total running time of the notebooks reduces by 1-25%. A large portion of the reduction can be attributed to faster data loading time (52-55% lower), as the size of the split dataset is 44% smaller.



Notebooks for NYC *dataset*	
Name	**Description of Operations**
NB1	Aggregations, filtering, adding features
NB2	Aggregations, filtering, adding features
NB3	Aggregations, adding features, count null/unique values, drop features, drop null values
Notebooks for FLIGHT *dataset*	
NB1	Aggregations, filtering, feature engineering
NB2	Aggregations, adding/dropping features

Figure 6: Notebooks for the NYC and FLIGHT datasets. When operating on split dataframes, we observe large reduction in peak memory usage ranging from 35-61%, while the overall running times of these notebooks reduces by 13-58%. A large portion of the reduction can be attributed to faster data loading time (36-41% lower for NYC dataset and 66-70% lower for FLIGHT dataset), as the size of the split dataset is 29% and 54% lower for the NYC and FLIGHT datasets respectively.

5 EVALUATION

In this section, we evaluate the performance of notebooks running on SplitDF (§5.1), followed by the efficiency of `SplitGen` for generating split data (§5.2), on the datasets listed in Table 1.

5.1 Running notebooks on SplitDF

To evaluate the effectiveness of split dataframes, we implemented ten Ibis notebooks spanning over three datasets (Table 1) – US Accidents (ACCIDENT), NYC parking tickets (NYC), and Flight status prediction (FLIGHT) with sizes ranging from 1.2GB to 4.8GB. These notebooks are re-implementations of top-voted notebooks on Kaggle for each of these datasets, and they cover the wide range of operations typically conducted for data analysis such as feature engineering, handling null values, and aggregations to name a few (see Fig. 5 and 6). These notebooks are available on GitHub [28], and the split datasets for these notebooks have been generated using the implementation of `SplitGen` in Velox (§4.2).

The results shown in Fig. 5 and 6 have been obtained by running the notebooks on a laptop machine with 16GB RAM, which is a popular environment for data scientists [2]. Memory usage reported in all experiments is the peak resident set size during the process' lifetime, obtained using the GNU `time` tool [31]. Overall, we find that the peak memory usage reduces by a significant amount of 19-61% when running on split dataframes compared to running on original dataframes.

Table 1: Top-voted CSV datasets from Kaggle. We analyze datasets with sizes ranging from 50MB to 4.8GB.

CSV Dataset Name	Size	#Attrs.
FIFA 20 complete player dataset ***(FIFA)*** [16]	51 MB	626
COVID-19 dataset ***(COVID)*** [10]	75 MB	77
Emergency - 911 Calls ***(911)*** [15]	123 MB	9
Brazilian E-Commerce Public Dataset by Olist ***(ECOMM)*** [9]	126 MB	52
Football Events ***(FBALL)*** [19]	183 MB	40
Data Science for Good: Kiva Crowdfunding ***(DSG)*** [14]	233 MB	54
515k Hotel Reviews in Europe ***(HOTEL)*** [5]	238 MB	17
Bitcoin Historical Data ***(BITCOIN)*** [8]	318 MB	8
FitBit Fitness Tracker Data ***(FITBIT)*** [17]	338 MB	259
US Accidents (2016-19) ***(ACCIDENT)*** [32]	1.2 GB	47
NYC Parking Tickets 2014 ***(NYC)*** [25]	1.9 GB	51
Flight Status Prediction (2018-19) ***(FLIGHT)*** [18]	4.8 GB	122

The running time of these notebooks reduces by 1-58%. A large portion of this reduction can be attributed to the reduction in data loading time, as the size of the split dataset is considerably smaller (44%, 29%, and 54% smaller for the ACCIDENT, NYC, and FLIGHT datasets respectively). The median reduction in data loading time for the three datasets is 52%, 41%, and 68% respectively. The running

time of the data analysis portion of the notebooks increases by 18-316%, as the queries now involve performing joins between the fact and dimension tables when operating on split dataframes. For each of the three datasets, the highest increase in data analysis time is observed for notebooks where a join involving all the dimension tables is performed (37%, 52% and 316% increase respectively for the ACCIDENT, NYC, and FLIGHT datasets).

Running the notebooks for the US Accidents dataset on a server machine with 160GB of RAM yielded a 55% median reduction in memory usage compared to the 22% median reduction we observe on the laptop machine with 16GB of RAM. While the exact reduction in memory usage is influenced by the behavior of the garbage collector in Python3 on different experimental setups, it is safe to say that operating on split dataframes results in significant reduction in memory usage. Larger RAM size on the server machine allowed experimenting with larger datasets as well. Running the FLIGHT NB1 (see Fig. 6) notebook on an expanded version of the Flight Delays and Cancellations dataset [18] of size 10GB yielded a net reduction in running time of 12%, and a net reduction in data loading time of 14% when running on split dataframes.

5.2 Splitting CSV Data with `SplitGen`

We perform splitting on twelve top-voted CSV datasets collected from Kaggle listed in Table 1. We chose a range of datasets with sizes ranging from 50MB to 4.8GB. We evaluate the reduction in size/memory arising from `SplitGen` in §5.2.1, followed by comparing `SplitGen` to dictionary encoding and normalization in §5.2.2.

5.2.1 Improvements in Size and Memory Utilization. To demonstrate the efficiency of the `SplitGen` algorithm, we compare the relative sizes of original vs split CSV datasets. Splitting is a one-time offline operation, executed on a server machine with 160GB of RAM. The time taken by `SplitGen` ranges from 1 minute for the smallest dataset (FIFA) to 38 minutes for the largest dataset (FLIGHT). Fig. 7 shows the reduction in raw data size from splitting – for six out of the twelve datasets tested, we obtain a substantial reduction in total size of more than 40% (median reduction of 39.5%), which shows the effectiveness of `SplitGen` in reducing redundancy.

To show the promise of working with split tabular data, we compare the memory footprint of three prominent libraries in the data science ecosystem – PyArrow [7] (Fig. 8a), DuckDB [56] (Fig. 8c), and Pandas [26] (Fig. 8b) – when loading the original vs the split CSV dataset on a laptop with 16GB RAM [2]. For the three libraries, we obtain a median reduction in memory usage of 39.0%, 33.5%, and 35.2% respectively. Note that for the FLIGHT dataset, both PyArrow and Pandas run out of memory when loading the original raw data, while Pandas runs out of memory for the NYC dataset as well. Splitting reduces the memory footprint, enabling Pandas and PyArrow to load these large datasets.

5.2.2 Comparison to Dictionary Encoding and Normalization. Given that the goal of splitting is to reduce the size of the data, splitting is also related to compression. Dictionary encoding [4] is a special case of splitting where individual attributes are split into dimension tables. We compared `SplitGen` to 1) naive, and 2) improved dictionary encoding. A naive implementation of dictionary encoding involved splitting each attribute of the dataset into a dimension table, and we found that this strategy *increases* the dataset size for

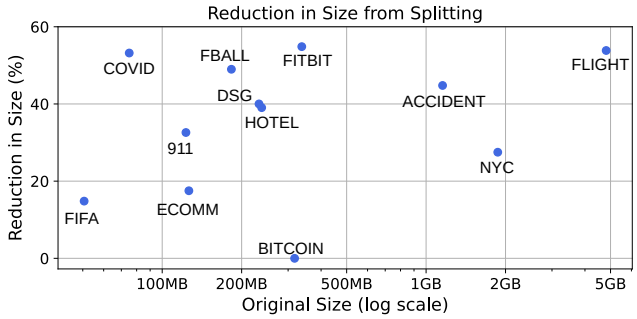


Figure 7: Reduction in CSV dataset size from splitting. For six of the twelve datasets, we obtain a large reduction in size of over 40% from splitting using the `SplitGen` algorithm.

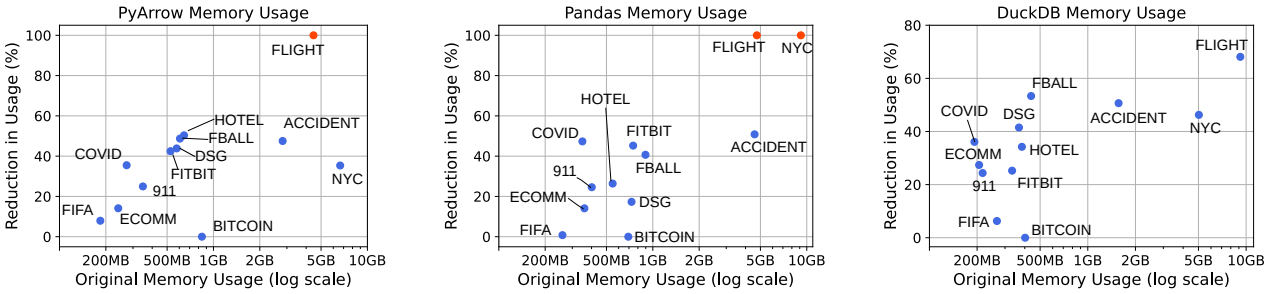
four out of the twelve datasets tested (Table 1). For rest of the eight datasets, we found that `SplitGen` yielded a 7-51% further reduction in dataset size compared to naive dictionary encoding. To improve the implementation of dictionary encoding, we modified `SplitGen` to only consider attribute groups of size one, i.e, each attribute is evaluated to estimate if dictionary encoding is likely to result in a net reduction in size. Compared to the improved implementation of dictionary encoding, `SplitGen` yields a 8-28% further reduction in dataset size for six out of the twelve datasets, while the performance is very similar (within 3%) for the rest of the datasets.

Normalization is yet another technique that has the dual goal of reducing redundancy and improving the integrity of the data. Normalization is expensive (noted to be $O(n^2 \frac{m}{2}^2 2^m)$ [48], i.e., quadratic in the number of rows n and exponential in the number of attributes m), and often involves manual intervention to obtain an effective schema. The only end-to-end automatic normalization tool we found was Metanome [51] which performs normalization to BCNF [38]. In practice, we found automatic normalization to be not very effective when it comes to reducing the size of the dataset, as in some cases the generated schema contained compound keys resulting in duplication of attributes across tables. For instance, the normalized schema for the Football Events (FBALL) dataset yielded a 130% increase in size of the dataset, unlike `SplitGen` that yields a 50% reduction in size for the FBALL dataset.

6 RELATED WORK

The issues related to high main memory usage of dataframe libraries such as Pandas [26, 60] are well known, making it challenging to analyze large datasets. Consequently, there have been multiple efforts in the community to scale data analysis to larger datasets. Modin [55] and Dask [3] leverage a distributed runtime such as Ray [49] to perform operations on distributed dataframes. Other efforts are Vaex [33] which uses lazy evaluation and memory mapping, and RAPIDS cuDF [12] which utilizes the GPU to enable data scientists to work with larger datasets on a single machine. The primary goal of these efforts remains scaling to larger datasets, and not necessarily optimizing memory usage [13, 30].

Inspired by pandas, newer dataframe APIs have been developed that run atop relational database systems to draw benefits off their performance and efficiency. The Ibis [20] API is supported over multiple backends ranging from in-process DBMS such as DuckDB [56], to cloud-based solutions such as Snowflake [39]. Grizzly [43] utilizes



(a) For six out of twelve datasets tested, we obtain over 40% reduction in memory usage when loading split CSV datasets. For the FLIGHT dataset (marked red, in which case the x-axis indicates memory usage for the split dataset), PyArrow ran out of memory when loading the original dataset.

(b) For six out of twelve datasets tested, we obtain over 40% reduction in memory usage when loading split CSV datasets. For the FLIGHT and NYC datasets (marked red, in which case the x-axis indicates memory usage for the split dataset), Pandas ran out of memory when loading the original datasets.

(c) For five out of twelve datasets tested, we obtain over 40% reduction in memory usage when loading split CSV datasets.

Figure 8: Reduction in memory usage of (a) PyArrow, (b) Pandas, and (c) DuckDB when loading from split vs original dataset.

a transpiler to convert pandas-like API to SQL. A similar approach is taken by the PyFroid compiler of Magpie [46], which converts pandas expressions into language agnostic IR utilizing Ibis, and leverages optimized relational backends in the cloud whenever possible. Other notable solutions that run atop relational DBMS are PySpark [27] and Koalas [23]. While these efforts recognize the performance and efficiency of relational DBMS, they do not leverage the fundamental optimization of lossless decomposition employed by relational DBMS.

There is a rich theory of normalization [36–38, 40] for removing redundancy and improving the integrity of relational databases. The key distinction between normalization and splitting is that normalization requires obtaining functional dependencies (FDs) from the data. Authors of [52] conduct a thorough evaluation of prominent functional dependency discovery algorithms. In general, FD discovery is an expensive operation and its complexity has been shown to be $O(n^2(\frac{m}{2})^2 2^m)$ [48], and different algorithms can broadly be classified based on finding 1) exact FDs, such as DFD [35], GORDIAN [59], and HyFD [53], and 2) approximate FDs (where a dependency can be violated by only a fraction of the rows) such as PYRO [47], FDX [61], and TANE [44]. Although obtaining functional dependencies from the data is strictly not required for splitting, functional dependencies can help generate attribute groups with correlated/dependent attributes grouped together to further improve the efficiency of splitting. Given the expensive nature of FD discovery, techniques involving sampling and discovering correlation between columns ("soft" FDs) such as CORDS [45] can potentially be used to improve upon `SplitGen`.

Splitting can be interpreted as a compression technique. Unlike syntactic compression techniques such as Lempel-Ziv encoding [62] that treat data as a string, splitting involves extracting structure out of the data by decomposing it into smaller tables. Splitting can be used in conjunction with techniques for compression, as well as techniques for efficient storage such as compressed columnar storage [4]. Factorization [50] is yet another technique used for reducing redundancy in relational database systems which involves representing query results as compact cartesian products, and is thus orthogonal to splitting.

7 CONCLUSIONS AND FUTURE WORK

Inspired by the practice of normalization in relational databases, in this work we introduced a technique called *splitting*. Splitting involves performing lossless-join decomposition by explicitly introducing joining keys, and thus does not require the user to perform functional dependency discovery and schema design. We propose splitting dataframes to reduce redundancy in the data, thus improving memory efficiency. We developed SplitDF, an implementation of split dataframes in Ibis, that makes minimal changes to the Ibis API providing the impression of a unified dataframe while operating on split data to improve memory efficiency. Choosing efficient splits is a critical aspect to reduce redundancy in the data, and we developed a greedy algorithm `SplitGen` that automatically produces efficient splits without requiring the data scientist to perform schema design. We implemented `SplitGen` in Velox to generate split CSV files.

For evaluating the effectiveness of splitting dataframes, we implemented ten notebooks spanning over the largest three datasets – US Accidents (1.2GB), NYC parking tickets (1.9GB), and Flight status prediction (4.9GB). We find that across all the notebooks, peak memory consumption reduces by 19-61%, while the running time reduces by 1-58%, showing the promise of splitting dataframes. We evaluated `SplitGen` on twelve top-voted CSV datasets from Kaggle, and found that for six out of twelve datasets, we obtain a substantial reduction of over 40% in size. Future work can involve implementing split dataframes in other libraries such as Pandas [26]. Splitting can also be implemented in distributed dataframe libraries such as Modin [55] and Dask [3]. One could explore the application of techniques such as vertical partitioning [34] to group together frequently co-accessed attributes while generating a split schema. Such a workload-aware approach can benefit not just distributed frameworks, but can also help improve the running time of notebooks on a single machine by reducing the number of joins required.

ACKNOWLEDGMENTS

This research was supported by the National Science Foundation under grant OAC-1835446 and a David DeWitt Fellowship.

REFERENCES

[1] 2010. How to add an auto-incrementing primary key to an existing table in PostgreSQL? https://stackoverflow.com/questions/2944499/how-to-add-an-auto-incrementing-primary-key-to-an-existing-table-in-postgresql.

[2] 2020. The Definitive Data Scientist Environment Setup. https://whiteboxml.com/blog/the-definitive-data-scientist-environment-setup.

[3] 2022. Dask. https://www.dask.org/.

[4] 2022. Lightweight Compression in DuckDB. https://duckdb.org/2022/10/28/lightweight-compression.html.

[5] 2023. 515k Hotel Reviews Data in Europe. https://www.kaggle.com/datasets/jiashenliu/515k-hotel-reviews-data-in-europe.

[6] 2023. Apache Arrow. https://arrow.apache.org/.

[7] 2023. Apache Arrow Python Bindings. https://arrow.apache.org/docs/python/index.html.

[8] 2023. Bitcoin Historical Data. https://www.kaggle.com/datasets/mczielinski/bitcoin-historical-data.

[9] 2023. Brazilian E-commerce Public Dataset by Olist. https://www.kaggle.com/datasets/olistbr/brazilian-ecommerce.

[10] 2023. COVID 19 Dataset. https://www.kaggle.com/datasets/imdevskp/corona-virus-report.

[11] 2023. Create Sequence in DuckDB. https://duckdb.org/docs/sql/statements/create_sequence.html.

[12] 2023. cuDF - GPU DataFrames. https://github.com/rapidsai/cudf.

[13] 2023. Dask Memory limits reached in simple ETL-like data transformations. https://dask.discourse.group/t/memory-limits-reached-in-simple-etl-like-data-transformations/1687.

[14] 2023. Data Science for Good – Kiva Crowdfunding. https://www.kaggle.com/datasets/kiva/data-science-for-good-kiva-crowdfunding.

[15] 2023. Emergency - 911 Calls. https://www.kaggle.com/datasets/mchirico/montcoalert.

[16] 2023. FIFA 20 complete player dataset. https://www.kaggle.com/datasets/stefanoleone992/fifa-20-complete-player-dataset.

[17] 2023. Fitbit Fitness Tracker Data. https://www.kaggle.com/datasets/arashnic/fitbit.

[18] 2023. Flight Status Prediction. https://www.kaggle.com/datasets/robikscube/flight-delay-dataset-20182022.

[19] 2023. Football Events. https://www.kaggle.com/datasets/secareanualin/football-events.

[20] 2023. The Ibis Project. https://ibis-project.org/.

[21] 2023. Importing Data in DuckDB. https://duckdb.org/docs/data/overview.html.

[22] 2023. Kaggle: Your Machine Learning and Data Science Community. https://www.kaggle.com/.

[23] 2023. Koalas. https://github.com/databricks/koalas.

[24] 2023. Lossless Join Decomposition. https://en.wikipedia.org/wiki/Lossless_join_decomposition.

[25] 2023. NYC Parking Tickets. https://www.kaggle.com/datasets/new-york-city/nyc-parking-tickets/.

[26] 2023. pandas. https://pandas.pydata.org/.

[27] 2023. PySpark Documentation. https://spark.apache.org/docs/latest/api/python.

[28] 2023. Splitting. https://github.com/UWQuickstep/splitting.

[29] 2023. The SQLGlot library. https://sqlglot.com/sqlglot.html.

[30] 2023. Tackling Excessive Memory Usage with Dask Dataframes from Parquet Files. https://saturncloud.io/blog/tackling-excessive-memory-usage-with-dask-dataframes-from-parquet-files/.

[31] 2023. time(1) - Linux Manual Page. https://man7.org/linux/man-pages/man1/time.1.html.

[32] 2023. US Accidents 2019. https://www.kaggle.com/datasets/sobhanmoosavi/us-accidents.

[33] 2023. Vaex.io: An ML Ready Fast Dataframe for Python. https://vaex.io/.

[34] 2023. What is data partitioning, and how to do it right. https://www.cockroachlabs.com/blog/what-is-data-partitioning-and-how-to-do-it-right/.

[35] Ziawasch Abedjan, Patrick Schulze, and Felix Naumann. 2014. DFD: Efficient Functional Dependency Discovery. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management* (Shanghai, China) *(CIKM '14)*. Association for Computing Machinery, New York, NY, USA, 949–958. https://doi.org/10.1145/2661829.2661884

[36] E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (jun 1970), 377–387. https://doi.org/10.1145/362384.362685

[37] E. F. Codd. 1971. Further Normalization of the Data Base Relational Model. *Research Report / RJ / IBM / San Jose, California* RJ909 (1971). https://api.semanticscholar.org/CorpusID:45071523

[38] E. F. Codd. 1974. Recent Investigations in Relational Data Base Systems. In *ACM Pacific*. https://api.semanticscholar.org/CorpusID:47325247

[39] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 215–226. https://doi.org/10.1145/2882903.2903741

[40] Hugh Darwen, C. J. Date, and Ronald Fagin. 2012. A Normal Form for Preventing Redundant Tuples in Relational Databases. In *Proceedings of the 15th International Conference on Database Theory* (Berlin, Germany) *(ICDT '12)*. Association for Computing Machinery, New York, NY, USA, 114–126. https://doi.org/10.1145/2274576.2274589

[41] Philippe Flajolet, Eric Fusy, Olivier Gandouet, and Frédéric Meunier. 2012. HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm. *Discrete Mathematics Theoretical Computer Science* DMTCS Proceedings vol. AH,... (03 2012). https://doi.org/10.46298/dmtcs.3545

[42] Philippe Flajolet and G. Nigel Martin. 1985. Probabilistic counting algorithms for data base applications. *J. Comput. System Sci.* 31, 2 (1985), 182–209. https://doi.org/10.1016/0022-0000(85)90041-8

[43] Stefan Hagedorn. 2020. When sweet and cute isn't enough anymore: Solving scalability issues in Python Pandas with Grizzly. In *Conference on Innovative Data Systems Research*. https://api.semanticscholar.org/CorpusID:213180298

[44] Yká Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. 1999. TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. *Comput. J.* 42, 2 (1999), 100–111. https://doi.org/10.1093/comjnl/42.2.100

[45] Ihab F. Ilyas, Volker Markl, Peter Haas, Paul Brown, and Ashraf Aboulnaga. 2004. CORDS: Automatic Discovery of Correlations and Soft Functional Dependencies. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (Paris, France) *(SIGMOD '04)*. Association for Computing Machinery, New York, NY, USA, 647–658. https://doi.org/10.1145/1007568.1007641

[46] Alekh Jindal, K. Venkatesh Emani, Maureen Daum, Olga Poppe, Brandon Haynes, Anna Pavlenko, Ayushi Gupta, Karthik Ramachandra, Carlo Curino, Andreas Mueller, Wentao Wu, and Hiren Patel. 2021. Magpie: Python at Speed and Scale using Cloud Backends. In *Conference on Innovative Data Systems Research*. https://api.semanticscholar.org/CorpusID:231782138

[47] Sebastian Kruse and Felix Naumann. 2018. Efficient Discovery of Approximate Dependencies. *Proc. VLDB Endow.* 11, 7 (mar 2018), 759–772. https://doi.org/10.14778/3192965.3192968

[48] Jixue Liu, Jiuyong Li, Chengfei Liu, and Yongfeng Chen. 2012. Discover Dependencies from Data—A Review. *IEEE Transactions on Knowledge and Data Engineering* 24, 2 (2012), 251–264. https://doi.org/10.1109/TKDE.2010.197

[49] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I. Jordan, and Ion Stoica. 2017. Ray: A Distributed Framework for Emerging AI Applications. *CoRR* abs/1712.05889 (2017). arXiv:1712.05889 http://arxiv.org/abs/1712.05889

[50] Dan Olteanu and Maximilian Schleich. 2016. Factorized Databases. *SIGMOD Rec.* 45, 2 (sep 2016), 5–16. https://doi.org/10.1145/3003665.3003667

[51] Thorsten Papenbrock, Tanja Bergmann, Moritz Finke, Jakob Zwiener, and Felix Naumann. 2015. Data Profiling with Metanome. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1860–1863. https://doi.org/10.14778/2824032.2824086

[52] Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. 2015. Functional Dependency Discovery: An Experimental Evaluation of Seven Algorithms. *Proc. VLDB Endow.* 8, 10 (jun 2015), 1082–1093. https://doi.org/10.14778/2794367.2794377

[53] Thorsten Papenbrock and Felix Naumann. 2016. A Hybrid Approach to Functional Dependency Discovery. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 821–833. https://doi.org/10.1145/2882903.2915203

[54] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: Meta's Unified Execution Engine. *Proc. VLDB Endow.* 15, 12 (aug 2022), 3372–3384. https://doi.org/10.14778/3554821.3554829

[55] Devin Petersohn, William W. Ma, Doris Jung Lin Lee, Stephen Macke, Doris Xin, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya G. Parameswaran. 2020. Towards Scalable Dataframe Systems. *CoRR* abs/2001.00888 (2020). arXiv:2001.00888 http://arxiv.org/abs/2001.00888

[56] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984. https://doi.org/10.1145/3299869.3320212

[57] Raghu Ramakrishnan and Johannes Gehrke. 2003. *Database management systems (3. ed.)*. McGraw-Hill.

[58] Phanwadee Sinthong and Michael J. Carey. 2021. PolyFrame: A Retargetable Query-Based Approach to Scaling Dataframes. *Proc. VLDB Endow.* 14, 11 (jul 2021), 2296–2304. https://doi.org/10.14778/3476249.3476281

[59] Yannis Sismanis, Paul G. Brown, Peter J. Haas, and Berthold Reinwald. 2006. GORDIAN: efficient and scalable discovery of composite keys. In *Very Large Data Bases Conference*. https://api.semanticscholar.org/CorpusID:16124888

[60] Wes McKinney. 2017. Apache Arrow and the "10 Things I Hate About pandas"'. https://wesmckinney.com/blog/apache-arrow-pandas-internals/.
[61] Yunjia Zhang, Zhihan Guo, and Theodoros Rekatsinas. 2020. A Statistical Perspective on Discovering Functional Dependencies in Noisy Data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 861–876. https://doi.org/10.1145/3318464.3389749
[62] J. Ziv and A. Lempel. 1978. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory* 24, 5 (1978), 530–536. https://doi.org/10.1109/TIT.1978.1055934