University of Milan

Faculty of Political, Economic and Social Sciences

Parallelized SON Algorithm for Frequent Itemset Mining on Large Datasets

Final Project in the *Algorithms for Massive Data* Module

Julia Maria Wdowinska

Data Science for Economics
II year
Master's Degree
Matriculation Number: 43288A

March 26, 2025

Contents

1 Introduction

Frequent itemset mining is an interesting task in data mining. The goal is to identify sets of items that frequently appear together in a dataset, enabling valuable insights such as co-purchase behavior in retail or user preferences in recommendation engines.

Numerous frequent itemset mining algorithms exist, each with its own advantages. This report presents an implementation of the SON algorithm (Savasere et al., 1995), where the Apriori algorithm (Agrawal and Srikant, 1994) is used for local itemset mining, and the solution is parallelized using Apache Spark. The methodology is applied to a dataset of Amazon book reviews (Bekheet, 2022) to identify books that are frequently *liked* together by users. Additionally, the implementation is tested for efficiency across varying dataset sizes, minimum support thresholds, and maximum itemset sizes.

The report is structured as follows: Section 2 provides an overview of the algorithms and their implementation. Section 3 introduces the dataset and preprocessing steps. Section 4 presents experimental results, including frequent itemset mining outcomes and scalability tests. Finally, Section 5 summarizes key findings and discusses potential future improvements.

2 Frequent Itemset Mining Implementation

2.1 Apriori Algorithm

The Apriori algorithm is a classical approach to frequent itemset mining. It takes as input a dataset of baskets - each represented as a set of items - and a minimum support threshold, which defines the percentage of baskets in which an itemset must appear to be considered frequent. The algorithm outputs all frequent itemsets of size 1 up to a specified maximum size k_{max}, as determined by the user. It works iteratively, scanning through baskets to count occurrences of itemsets of size k, where $k = 1, \ldots, k_{\max}$, and filtering them based on the support threshold.

Given that the number of possible item combinations grows rapidly with dataset size, the Apriori algorithm exploits the property that if an itemset is frequent, all of its subsets must also be frequent. Thus, an itemset of size k can only be frequent if all of its $(k-1)$-sized subsets are frequent. Leveraging this property, the algorithm prunes the search space by generating new candidate itemsets exclusively from previously discovered frequent itemsets, rather than considering all possible combinations. This approach improves efficiency by reducing both computational time and memory usage, as infrequent itemsets are eliminated early, avoiding unnecessary counting and storage.

The implementation of the Apriori algorithm for this project is structured as follows.

apriori Function:

The main function, `apriori`, takes three inputs:

- `baskets`: a list of sets, where each set represents a basket of items,

- `min_count`: the minimum count threshold, computed by multiplying the support threshold by the number of baskets being processed and truncating the decimal part to obtain an integer,
- `k_max`: the maximum itemset size to be found.

The function starts by iterating through baskets, adding each item as a key to dictionary `item_counts`, and incrementing its count upon each occurrence. This approach is memory-efficient, as only seen items are stored. Once all singletons are counted, the function retains those that meet `min_count` and stores them as follows:

- A set (`frequent_items`), used for filtering baskets in the next iteration.
- A list of tuples $((\text{item},),1)$ (`output_frequent_itemsets`), which forms part of the final output.

The function then iterates through k from 2 to `k_max` and, for each basket, it:

1. Filters the basket to retain only items in `frequent_items`, creating `filtered_basket`.
2. Generates candidate itemsets as follows:
   - For $k = 2$, all possible pairs of items from `filtered_basket` are created.
   - For $k > 2$, a list of subsets of size $k-1$ is generated from `filtered_basket`, retaining only those that were found to be frequent in the previous iteration. This list, called `frequent_subsets`, is then used to generate candidate itemsets of size k via the `generate_candidate_itemsets` function.
3. Updates dictionary `counts`, incrementing the count for each candidate itemset.

To avoid unnecessary computations, the function skips a basket:

1. If `filtered_basket` contains fewer than k items, as no candidate itemsets of size k can be formed.
2. For $k > 2$, if the number of subsets generated from `filtered_basket` is less than 2, as at least 2 subsets of size $k-1$ are required to form a candidate itemset of size k.

After counting candidate itemsets, those meeting `min_count` are stored in three formats:

- A set of tuples (itemset) (`frequent_itemsets`), used for filtering subsets in the next iteration.
- A set of individual items that appear in `frequent_itemsets` (`frequent_items`), used for filtering baskets in the next iteration.
- A list of tuples $((\text{itemset}),1)$, appended to `output_frequent_itemsets` for the final output.

The function terminates when it reaches k_{max} or when no frequent itemsets of size k are found in an iteration, and returns `output_frequent_itemsets`.

generate_candidate_itemsets Function:

The helper function, `generate_candidate_itemsets`, takes the following inputs:

- `frequent_subsets`: a list of tuples representing subsets of size $k-1$ generated from `filtered_basket`,
- `k`: the target size of candidate itemsets,
- `frequent_itemsets`: a set of tuples representing itemsets found to be frequent in iteration $k-1$.

The function iterates through all possible pairs of subsets in `frequent_subsets`. When a pair shares exactly $k-2$ elements, their union forms a candidate itemset of size k. This candidate is then added to the set of candidate itemsets only if all of its subsets of size $k-1$ exist in `frequent_itemsets`. This pruning step prevents the inclusion of itemsets where two subsets (used to generate it) are frequent but at least one of the other subsets is not. The function returns a list of candidate itemsets.

2.2 SON Algorithm

The SON algorithm, introduced by Savasere et al. (1995), improves upon Apriori by first performing local frequent itemset mining on dataset partitions before globally verifying the candidates. It operates in two phases:

- **Phase I**: The dataset is partitioned into chunks, and each chunk is processed independently using a frequent itemset mining algorithm (e.g., Apriori) to identify locally frequent itemsets. The union of these locally frequent itemsets across all chunks forms the candidate set for the second phase.
- **Phase II**: The occurrences of each itemset in the candidate set are counted across the entire dataset, and only those that meet the global minimum support threshold are retained as frequent itemsets.

The key insight behind SON is that if an itemset is globally frequent, it is also frequent in at least one chunk. This property allows independent processing of each chunk while ensuring that all truly frequent itemsets are included in the candidate set identified in the first phase. As a result, the algorithm guarantees no false negatives. However, some false positives may arise in the first phase, meaning that some locally frequent itemsets may not be frequent globally. These are filtered out in the subsequent phase when counting is performed across the entire dataset.

The ability to independently handle chunks makes SON well-suited for parallelization. Its two-phase structure can be naturally expressed using the MapReduce framework and has been implemented in Apache Spark, as detailed below.

The input dataset is converted to a Resilient Distributed Dataset (RDD), meaning it is partitioned and distributed across multiple nodes, enabling parallel processing. It undergoes two MapReduce phases:

- **I Map Phase**: Each node processes its assigned partition of baskets independently, applying the `apriori` function to identify locally frequent itemsets. The output is a list of key-value pairs $((\text{itemset}), 1)$.
- **I Reduce Phase**: Locally frequent itemsets from all partitions are aggregated by grouping tuples by key and retaining only unique keys (i.e., itemsets). This step identifies the set of candidate itemsets, i.e., those that are frequent in at least one partition.
- **II Map Phase**: Each node processes its assigned partition of baskets, counting occurrences of candidate itemsets using the `count_occurrences` function. The output is a list of key-value pairs $((\text{itemset}), c)$, where c represents the local count of the itemset.
- **II Reduce Phase**: Counts from all partitions are aggregated by grouping tuples by key and summing the associated counts. Only itemsets that meet the global minimum support threshold are retained, producing the final output as a list of key-value pairs $((\text{itemset}), g)$, where g is the global count of the itemset.

count_occurrences Function:

The `count_occurrences` function takes the following inputs:

- `baskets`: a list of sets, where each set represents a basket of items,
- `itemsets`: a list of tuples, where each tuple represents a candidate itemset.

The function iterates over each itemset in `itemsets` and checks whether it appears in any basket in `baskets`. If so, it is added to dictionary `counts` (if not already present), and its count is incremented. The function returns a list of tuples $((\text{itemset}), c)$, where c represents the total count of the itemset.

2.3 Association Rule Mining

Frequent itemset mining is often just the first step in discovering meaningful patterns in data. A more insightful way to present the results is through *association rules*, which take the form $I \to j$, where I is a set of items and j is a single item. This rule implies that if all items in I appear in a basket, then j is also *likely* to be present.

To quantify this likelihood, the *support* of an itemset I is defined as the fraction of baskets in which I appears. The *confidence* of an association rule $I \to j$ is then given by the ratio of the support of $I \cup \{j\}$ to the support of I.

Another important measure is the *interest* of an association rule, defined as the difference between its confidence and the support of j. This measure is crucial because even if a rule has nonzero confidence, it does not necessarily indicate a meaningful relationship. If the fraction of baskets containing I that also contain j is the same as the overall fraction of baskets containing j, then I has no influence on j, resulting in an interest value of zero.

For an association rule to be considered important, both confidence and interest should be positive, meaning the presence of I increases the likelihood of j. However, negative interest can also be informative, as it highlights cases where the presence of I discourages the presence of j, revealing inverse relationships in the data.

association_rules Function:

The `association_rules` function takes the following inputs:

- `frequent_itemsets`: a list of tuples $((\text{itemset}), g)$, where g is the global count of the itemset,
- `n_baskets`: the total number of baskets,
- `min_confidence`: the minimum confidence threshold for an association rule to be included in the output (the default is 0.0).

The function first converts `frequent_itemsets` into a dictionary, storing itemsets as keys and their counts as values for efficient lookup. For each itemset of size k where $k > 1$, all possible subsets of size $k-1$ are generated. Each subset is considered as the antecedent (left-hand side) of an association rule, while the remaining element forms the consequent (right-hand side).

The support, confidence, and interest of each rule are computed according to the definitions provided earlier. An association rule is added to the output list if its confidence meets the `min_confidence` threshold. The result is returned as a data frame with columns: *antecedent*, *consequent*, *support* (i.e., support of the union of antecedent and consequent), *confidence*, and *interest*.

3 Dataset Description and Preprocessing

The dataset, available at `https://www.kaggle.com/datasets/mohamedbakhet/amazon-books-reviews`, contains 3,000,000 reviews of books purchased through Amazon. These reviews were written by 1,008,973 users and cover 221,998 unique books. An overview of the dataset variables is provided in Table 1.

Table 1: Amazon Reviews Dataset Variables

Variable	**Data Type**	**Description**
Id	string	Book ID
Title	string	Book title
Price	float	Book price
User_id	string	User ID
profileName	string	User name
review/helpfulness	string	Review helpfulness rating
review/score	float	Rating
review/time	integer	Review time
review/summary	string	Review text summary
review/text	string	Review text

A straightforward approach to frequent itemset mining on this dataset would be to identify books that were frequently reviewed together. However, since the dataset contains more information than just which books were reviewed, a different perspective was considered. To take advantage of this additional information, the following assumption was introduced:

A user is considered to have liked a book if they rated it 4 or higher.

Based on this assumption, the frequent itemset mining problem was reframed to focus on identifying books that were frequently *liked* together.

To align with this new formulation, the dataset was refined by renaming variables as follows: Id → book_id, Title → title, User_id → user_id, and review/score → rating. Only these four variables were retained for further analysis. Two of them were found to contain missing values, as summarized in Table 2.

Table 2: Missing Values Count

Variable	**Count**
book_id	0
title	208
user_id	561 787
rating	0

Since a missing user_id value prevents associating the book with other books rated by the same user, all reviews without a user_id were excluded. Additionally, reviews with a missing book title, accounting for only 0.007% of the data, were dropped. After these steps, the dataset was reduced to 2,438,018 rows.

Next, reviews were filtered to include only those with a rating of 4 or higher, thereby considering only books that users *liked*. This reduced the dataset to 1,954,329 rows.

It was then discovered that some book titles were associated with two or more different book IDs. To resolve this, only unique title-user_id pairs were kept. This had two effects:

1. If user X *liked* book A under multiple book IDs, only one pair user X-book A was retained.
2. If user X *liked* book A multiple times under the same book ID, only one pair user X-book A was kept.

Both of these effects were intended, as for the purpose of frequent itemset mining, the focus is on the association between a user and a book, rather than the number of times this relationship occurs. After this step, the dataset was reduced to 1,690,999 rows. Since book_id was no longer accurate, it was replaced with a new integer-based book_id for each unique book title. Additionally, user_id was converted to an integer to save memory.

The remaining columns were renamed as follows: user_id → basket_id, and book_id → item_id. This renaming was done to make the analysis more universal by treating the dataset in terms of baskets and items, regardless of their specific context (e.g., users and books).

Baskets containing only one item were excluded, as they cannot contribute to itemsets of size greater than 1, and association rule mining requires frequent itemsets of at least size 2. This reduced the dataset to 1,076,095 rows. Finally, the dataset was converted into an RDD, with baskets formed by grouping items by basket_id and returning them as sets. The total number of baskets created was 221,587.

4 Experimental Results

4.1 Frequent Pair Mining Results

The SON algorithm was applied to this collection of baskets. The minimum support threshold was set to 0.01%, as higher thresholds resulted in very few frequent itemsets. Typically, the support threshold is set much higher, but this dataset had a particular characteristic: a large number of unique books, each *liked* by only a few users. This could partly reflect the dataset's nature. However, it was also observed that the same book often appeared under multiple slightly different titles, such as *The Hobbit or There and Back Again* and *The Hobbit; Or, There and Back Again*. As a result, these variations were treated as distinct books, leading to separate support counts rather than being aggregated.

The code was executed on Google Colab, a cloud-based platform providing 12.67 GB of RAM and 2 virtual CPU cores. The number of partitions was set to 2, as testing with 4, 8, 16, 32, and 64 partitions led to increased execution time. Based on the number of baskets in each partition and the entire dataset, along with the chosen support threshold, the local and global minimum count thresholds were calculated. The maximum itemset size `k_max` was set to 2. The execution time was 244.04 seconds (approx. 4 minutes and 4 seconds), identifying 7,289 frequent singletons and 9,565 frequent pairs.

Following this, association rule mining was performed with no minimum confidence threshold to retrieve all possible association rules, resulting in 19,130 rules. Rules with negative or zero interest were filtered out, as the focus was on identifying books whose presence among a user's *liked* books increases the likelihood of another book being present. Additionally, rules with confidence of 0.8 or higher were discarded, as further examination revealed that highly confident rules often involved the same book with slight title variations - an issue previously noted. After applying these filters, 10,500 association rules remained.

Finally, a simple recommendation system was implemented. For a book_id randomly selected from the data, the consequent from the association rule with the highest interest was retrieved. The result was then printed in the following format: "Readers who enjoyed *Emma (Summer Classics)* often also liked *Pride & Prejudice (New Windmill)*."

4.2 Testing Implementation Scalability

Beyond mining frequent pairs using a fixed minimum support threshold, scalability tests were conducted to assess how the implementation performs under varying dataset sizes, minimum support thresholds, and maximum itemset sizes. Execution time was used as an indicator of implementation efficiency.

4.2.1 Effect of Dataset Size on Execution Time

The first test examined execution time as set of baskets increased. Since no additional data beyond the 221,587 baskets was available, artificial data subsets were created through sampling, either without or with replacement, depending on whether the sampling fraction was less or greater than 1. The following fractions were considered: 0.1, 0.2, 0.4, 0.8, 1.6, 3.2, and 6.4. The minimum support threshold was set to 0.1%, and `k_max` to 2.

Figure 1 shows the results. While the number of baskets approximately doubles at each step, execution time increases by less than a factor of 2 in most cases - for instance, when moving from 709,750 to 1,419,286 baskets, the increase is 1.57. The only exception occurs when increasing from 44,506 to 88,514 baskets, where execution

time grows by slightly more than 2. This sublinear scalability suggests that the implementation benefits from the two-phase approach of the SON algorithm and Apache Spark's parallelization, enabling efficient processing of large datasets.

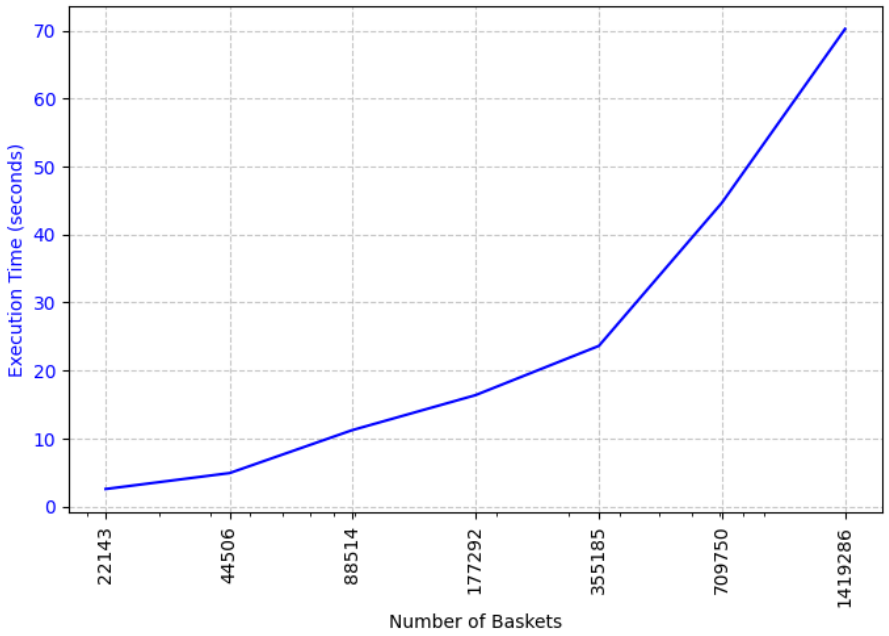

Figure 1: Number of Baskets vs. Execution Time

4.2.2 Effect of Minimum Support (min_support) on Execution Time

The second test examined execution time as the minimum support threshold decreased. The entire dataset was used, with `k_max` held constant at 2. The tested `min_support` values were 0.01, 0.005, 0.0025, 0.00125, 0.000625, 0.0003125, 0.00015625, and 0.000078125, each halving the previous threshold.

Results presented in Figure 2 reveal different scalability patterns based on the number of frequent itemsets found:

- For small frequent itemset counts (up to 1,226 at `min_support` = 0.00125), execution time scales sublinearly.
- For medium frequent itemset counts (up to 9,504 at `min_support` = 0.00015625), execution time scales approximately linearly.
- For large frequent itemset counts (above 25,000 at `min_support` = 0.000078125), execution time exhibits superlinear growth.

Although execution time shows superlinear growth for the largest number of frequent itemsets found, the total runtime remains manageable at 459.17 seconds (approx. 7 minutes and 39 seconds). This further demonstrates the implementation's ability to efficiently handle large datasets, successfully processing around 200,000 baskets while identifying a substantial number of frequent itemsets, including singletons and pairs, within 8 minutes.

4.2.3 Effect of Maximum Itemset Size (k_max) on Execution Time

The final test examined execution time for different values of `k_max` while using the full dataset and a fixed minimum support threshold of 0.1%. The tested values of `k_max` were 2, 3, 4, and 5.

Results presented in Figure 3 indicate that execution time increases approximately linearly with the number of frequent itemsets found when `k_max` increases from 2 to 3. Beyond this point, execution time grows slightly superlinearly, with the increase for `k_max` = 5 being 1.5 times the increase in frequent itemsets found.

Despite this, finding frequent 4- and 5-itemsets remains computationally feasible, as even in the largest case, where nearly 9,000 frequent itemsets, including 5-itemsets, are found, execution time remains under 4 minutes.

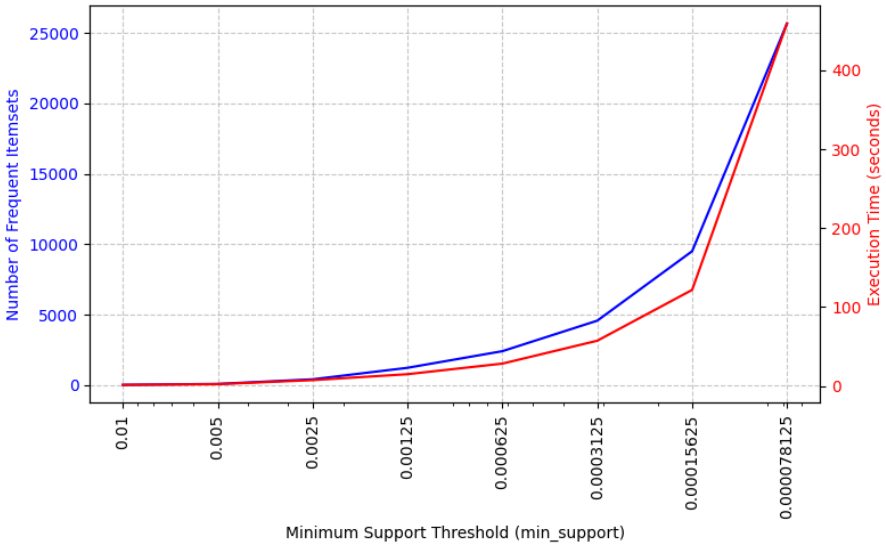

Figure 2: Minimum Support Threshold vs. Execution Time

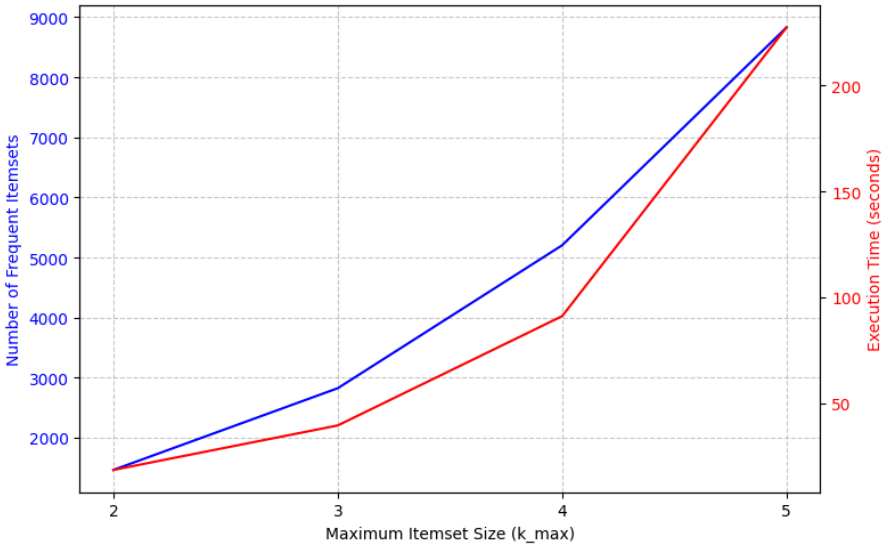

Figure 3: Maximum Itemset Size vs. Execution Time

5 Conclusions

This project successfully implemented a parallelized version of the SON algorithm using Apache Spark, with the Apriori algorithm employed for local itemset mining within partitions. The implementation was tested on a dataset of Amazon book reviews to evaluate its effectiveness and scalability.

Mining for frequent pairs in a collection of 221,587 baskets with a support threshold of 0.01% took approximately 4 minutes and identified 16,854 frequent itemsets. A subsequent association rule mining step allowed for the discovery of 10,500 rules with positive interest, enabling the creation of a simple recommendation system. In this system, for each book purchased by a user, another book could be suggested in the form of a banner: "Other users often also liked ...".

Further scalability tests demonstrated that execution time scales sublinearly with dataset size, indicating efficient parallelization. When varying the minimum support threshold, execution time remained manageable even when identifying a large number of frequent itemsets, demonstrating the robustness of the approach. However, for extremely low support values, execution time increased slightly superlinearly due to the exponential growth in frequent itemsets. Similarly, increasing `k_max` resulted in an approximately linear increase in execution time for small itemsets, while higher values led to a slightly superlinear increase. Despite this, discovering frequent 4- and 5-itemsets remained computationally feasible.

Overall, the implementation scales well to large datasets and maintains efficiency across different parameter settings. Future work could aim at optimizing memory usage, dynamically adjusting partitioning strategies, and

integrating more advanced pruning techniques to further improve execution time for low support thresholds and large itemset sizes. Additionally, applying the algorithm to diverse datasets could provide further insights into its performance across various domains.

References

Agrawal, R. and Srikant, R. (1994). Fast Algorithms for Mining Association Rules. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, pages 487–499.

Bekheet, M. (2022). *Amazon Books Reviews* [Data set]. `https://www.kaggle.com/datasets/mohamedbakhet/amazon-books-reviews`.

Leskovec, J., Rajaraman, A., and Ullman, J. D. (2020). *Mining of Massive Datasets*. Cambridge University Press, 3rd edition.

Savasere, A., Omiecinski, E., and Navathe, S. (1995). An Efficient Algorithm for Mining Association Rules in Large Databases. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB)*, pages 432–444.