**Содержание**

[Введение 5](#_Toc532952188)

[1. Научно-исследовательская часть 7](#_Toc532952189)

[1.1 Анализ предметной области 7](#_Toc532952190)

[1.1.1 Фискальные данные 7](#_Toc532952191)

[1.1.2 Интеллектуальный анализ данных 8](#_Toc532952192)

[1.1.3 Анализ существующих решений 10](#_Toc532952193)

[1.2 Аналитика фискальных данных 12](#_Toc532952194)

[1.2.1 Постановка задачи 12](#_Toc532952195)

[1.2.2 Подходы к анализу фискальных данных 15](#_Toc532952196)

[1.2.3 Алгоритм BRUTEFORCE 15](#_Toc532952197)

[1.2.4 Алгоритм Apriori 17](#_Toc532952198)

[1.2.5 Алгоритм Eclat 18](#_Toc532952199)

[1.2.6 Алгоритм Frequent Pattern-Growth Strategy (FPG) 20](#_Toc532952200)

[1.2.7 Поиск оптимального подхода к поиску ассоциативных правил 26](#_Toc532952201)

[1.3 Выводы 27](#_Toc532952202)

[2. Проектно-конструкторская часть 29](#_Toc532952203)

[2.1 Функциональные требования 29](#_Toc532952204)

[2.2 Нефункциональные требования 30](#_Toc532952205)

[2.3 Архитектура системы 30](#_Toc532952206)

[2.4 Проектирование базы данных системы 33](#_Toc532952207)

[2.4.1 Концептуальное проектирование БД 33](#_Toc532952208)

[2.4.2 Физическая модель данных 34](#_Toc532952209)

[2.5 Проектирование решения 36](#_Toc532952210)

[3. Проектно-технологическая часть 38](#_Toc532952211)

[3.1 Платформа для разработки системы 38](#_Toc532952212)

[3.2 Разработка модуля анализа 39](#_Toc532952213)

[3.3 Регистрация транзакций в системе 40](#_Toc532952214)

[3.4 Описание работы системы 42](#_Toc532952215)

[3.4.1 Поиск источника данных 42](#_Toc532952216)

[3.5 Тестирование системы 43](#_Toc532952217)

[3.6 Развертывание системы 43](#_Toc532952218)

[Заключение 46](#_Toc532952219)

[Список использованных источников 47](#_Toc532952220)

[Список литературы 47](#_Toc532952221)

[ПРИЛОЖЕНИЕ А 49](#_Toc532952222)

# Введение

В настоящее время вопросы интеллектуального анализа данных (Data Mining) становятся все более актуальными. Это связано с тем, что с развитием информационных технологий процессы сбора и хранения различных данных стали существенно проще и дешевле. Для того, чтобы извлекать из накопленных данных полезную информацию, необходимо применять определенные алгоритмы обработки информации. Разработка таких алгоритмов требует значительных интеллектуальных усилий.

Операторы фискальных данных (ОФД) также предоставляют своим клиентам инструменты анализа фискальных данных. Однако перечень таких инструментов в большинстве случаев ограничивается просмотром объемов выручки, прошедших через ОФД, а также просмотра общей статистики по конкретным кассам (выручка, количество чеков и т.п.).

Из вышесказанного можно сделать вывод, что данные, которыми обладают ОФД используются недостаточно эффективно. Повысить эффективность их использования можно за счет применения современных методов интеллектуального анализа данных. Например, можно строить математические модели, позволяющие прогнозировать какие-либо события (изменения спроса на определенный товар, отток покупателей и т.п.), или же выявлять в данных различные нетривиальные паттерны (в каком районе выше спрос на определенные товары, или наличие каких товаров влияет на спрос другого товара). Такая информация может помочь клиентам ОФД повысить эффективность своих продаж, и как следствие получить дополнительную прибыль.

Цель данной работы заключается в исследовании методов интеллектуального анализа данных в приложении к аналитике фискальных данных с последующим применением этих методов в разработке аналитической системы.

Задачи, которые необходимо выполнить для достижения поставленной цели следующие:

* Провести анализ предметной области;
* Сформулировать задачу анализа фискальных данных с точки зрения бизнеса;
* Сформулировать задачу анализа фискальных данных с математической точки зрения;
* Разработать аналитический модуль системы;
* Реализовать возможность интеграции системы с другими системами;

# Научно-исследовательская часть

## Анализ предметной области

* + 1. Фискальные данные

Фискальные данные - сведения о расчетах, в том числе сведения об организации или индивидуальном предпринимателе, осуществляющих расчеты, о контрольно-кассовой технике, применяемой при осуществлении расчетов, и иные сведения, сформированные контрольно-кассовой техникой или оператором фискальных данных [1]. Другими словами, фискальные данные – информация, содержащаяся в чеках, генерируемых ККТ при выполнении оплаты (возврата) каких-либо товаров или услуг.

Далее приведен перечень обязательных реквизитов кассового чека:

* Применяемая организацией система налогообложения;
* Признак расчета: продажа/возврат;
* Заводской номер фискального накопителя;
* Порядковый номер фискальных данных;
* Фискальный признак данных;
* Список приобретенных товаров со стоимостью, ценой и примененными скидками;
* НДС для каждой позиции;
* Заводской номер фискального накопителя;
* Номер фискального документа;
* Код фискальной передачи данных;
* Сумма НДС и ставка налога;
* Название ОФД;
* Адрес сайта ОФД;
* QR-код, служащий дополнительной проверкой подлинности чека [2].

Оператор фискальных данных (ОФД), принимает, обрабатывает, хранит и отправляет в Федеральную налоговую службу фискальные данные, передаваемые с контрольно-кассовой техники. Схема работы оператора фискальных данных приведена на рисунке 1.1.1.



Рисунок 1.1.1 – Схема работы ОФД

Из вышесказанного можно сделать вывод, что фискальные данные являются потенциальным источником ценной информации, а ОФД, имея доступ к этим данным, кроме функции передачи этих данных в ФНС может производить их анализ, с целью получения ценной информации. Для извлечения ценной информации из большого количества данных, необходимо использовать инструменты интеллектуального анализа данных (Data Mining).

* + 1. Интеллектуальный анализ данных

Интеллектуальный анализ данных (Data Mining) ‒ это процесс обнаружения в “сырых” данных ранее неизвестных, нетривиальных, практически полезных, доступных интерпретации знаний, необходимых для принятия решений в различных сферах человеческой деятельности.

В широком смысле это современная концепция анализа данных, предполагающая, что

* данные могут быть неточными, неполными (содержать пропуски), противоречивыми, разнородными, косвенными, и при этом иметь гигантские объёмы, поэтому понимание данных в конкретных приложениях требует значительных интеллектуальных усилий;
* сами алгоритмы анализа данных могут обладать «элементами интеллекта», в частности, способностью обучаться по прецедентам, то есть делать общие выводы на основе частных наблюдений; разработка таких алгоритмов также требует значительных интеллектуальных усилий;
* процессы переработки “сырых” данных в информацию, а информации в знания уже не могут быть выполнены по старинке вручную, и требуют нетривиальной автоматизации.

По составу решаемых задач Data Mining практически не отличается от стандартного набора средств, применяемых с середины XX века в области прикладной статистики. Основное различие заключается в эффективности алгоритмов и технологичности их применения. Подавляющее большинство классических процедур имеют время выполнения, квадратичное или даже кубическое по объёму исходных данных. При количестве объектов, превосходящем несколько десятков тысяч, они работают неприемлемо медленно даже на самых современных компьютерах. За последние десятилетия значительные усилия в области Data Mining были направлены на создание специализированных алгоритмов, способных выполнять те же задачи за линейное или даже логарифмическое время без существенной потери точности.

Сфера применения Data Mining ничем не ограничена, он применим в любых сферах человеческой деятельности, в которых существуют статистические данные: в розничной торговле, банковском деле, телекоммуникации, страховании, медицина, молекулярной генетике, прикладной химии и т.д.

Поиск ассоциативных правил является одним из самых популярных приложений Data Mining. Суть задачи заключается в определении часто встречающихся наборов объектов в большом множестве таких наборов [3].

* + 1. Анализ существующих решений

На текущий момент нет специализированных решений для анализа фискальных данных. Однако такие инструменты разрабатываются самими компаниями, предоставляющими услуги ОФД. Самым развитым в данном направлении можно считать оператор фискальных данных Яндекс.ОФД, поскольку у данной компании имеется богатый опыт в решении задач в области анализа данных. Еще одним преимуществом Яндекс.ОФД является то, что в своей аналитике в зависимости от задачи могут использоваться другие данные сервисов «Яндекса» «Погода», «Карты», «Пробки», «Поисковые запросы» и т. д. Погода может влиять на продажи определенных видов товаров, а пробки и дорожная ситуация влияют на трафик торговых точек [4].

Другим путем решения задачи анализа накопленных данных, является использования специализированных программных продуктов, позволяющих проводить аналитику различных данных. Одним из таких продуктов является система анализа данных Tableau. Данная система позволяет работать с различными типами данных и решать различные аналитические задачи. Также данная система предоставляет большое количество инструментов визуализации данных, а также просматривать их в зависимости от заданных показателей, например можно оценить состояние данных в выбранном временном срезе и т.п. Пример визуализации данных в Tableau представлен на рисунке 1.1.2.

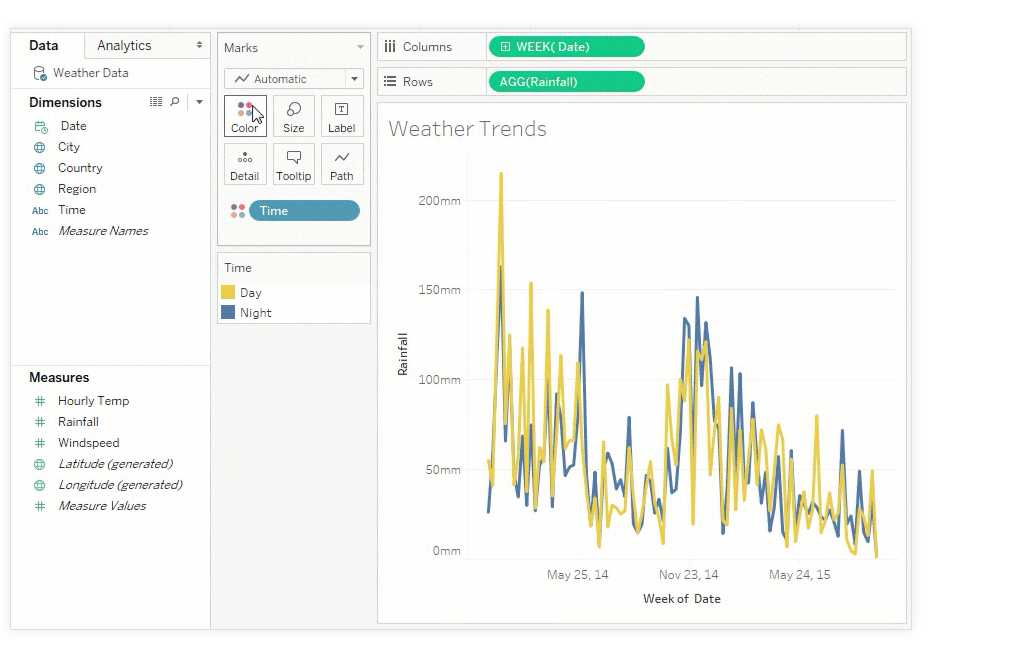


Рисунок 1.1.2 – Визуализация данных в системе Tableau

К недостаткам подхода использования специализированных программных продуктов можно отнести то, что внедрение и эксплуатация таких систем может быть достаточно затратным процессом. Также необходимо учитывать, что работа в такой системе требует от пользователя специальных знаний.

Таким образом разработка специализированных инструментов анализа фискальных данных является актуальной и коммерчески обоснованной задачей.

## Аналитика фискальных данных

* + 1. Постановка задачи

С точки зрения бизнеса задача может быть сформулирована следующим образом: на основании накопленных за определенный период фискальных данных, провести анализ рыночной корзины, а также представить результаты анализа в понятной для пользователя форме. Под анализом рыночной корзины здесь понимается процесс поиска наиболее типичных шаблонов покупок в супермаркетах путем анализа баз данных транзакций с целью определения комбинаций товаров, связанных между собой [5]. Полученные в ходе анализа рыночной корзины результаты могут быть использованы для решения следующих задач:

* оптимизация ассортимента и его размещения в торговых залах;
* повышение эффективности управления запасами;
* увеличение объемов продаж за счет предложения клиентам сопутствующих товаров;
* оценка эффективности различных рекламных кампаний (промо-акций);
* формирование персональных рекомендаций.

С точки зрения интеллектуального анализа данных, задача анализа рыночной корзины является задачей поиска ассоциативных правил [6].

Далее приведена математическая формулировка поставленной задачи.

Пусть – множество значений, называемых *элементами*. Множество называется *набором*. Набор, мощности (или размера) *k* называется k-набором. Далее, через обозначим множество всех k-наборов, то есть подмножеств с размером k.

Пусть – множество других элементов, называемых *идентификаторами транзакций* или *TIDs*. Множество называется *набором идентификаторов транзакций* (*TID-набором*). Предположим, что наборы и TID-наборыхранятся в упорядоченном виде.

*Транзакцией* называется кортеж формы , где – уникальный идентификатор транзакции, а *X* - набор элементов. Набор транзакций может обозначать набор всех покупок клиентов в супермаркете. При необходимости можно ссылаться на транзакцию по ее идентификатору *t*.

*Бинарная база данных* *D* представляет собой отношение между TID-набором и набором элементов в виде .

На рисунке 1.2.1 (б) приведен пример двоичной базы данных, для базы данных транзакций 1.2.1 (а). Здесь и В двоичной базе данных ячейка в строке t и столбце x равна 1 тогда и только тогда и 0 в противном случае.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | t | X | | 1 | ABDE | | 2 | BCE | | 3 | ABDE | | 4 | ABCE | | 5 | ABCDE | | 6 | BCD | | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | TID | A | B | C | D | E | | 1 | 1 | 1 | 0 | 1 | 1 | | 2 | 0 | 1 | 1 | 0 | 1 | | 3 | 1 | 1 | 0 | 1 | 1 | | 4 | 1 | 1 | 1 | 0 | 1 | | 5 | 1 | 1 | 1 | 1 | 1 | | 6 | 0 | 1 | 1 | 1 | 0 | |
| (а) | (б) |

Рисунок 1.2.1 – Представление базы данных транзакций

а – БД транзакций, б – бинарная БД

*Поддержка* *набора* *элементов* *X* в базе данных *D*, представляет собой количество транзакций в *D*, которые содержат *X* и рассчитывается по формуле (1.1).

(1.1)

*Относительная поддержка* *X* - это доля транзакций, которые содержат *X*, рассчитывается по формуле (1.2).

(1.2)

Считается, что набор элементов *X* является *частым* в *D*, если, где – минимальный порог поддержки, устанавливаемый пользователем.

*Правило ассоциации* представляет собой выражение , где *X* и *Y* являются не пересекающимися наборами, то есть и

*Поддержкой* *правила* является количество транзакций, в которых одновременно находятся оба набора X, и Y, рассчитывается по формуле (1.3).

(1.3)

*Относительная поддержка правила* определяется как доля транзакций, в которых одновременно находятся оба набора X, и Y, рассчитывается по формуле (1.4).

(1.4)

*Достоверность правила* - условная вероятность того, что транзакция содержит Y, учитывая, что она содержит X, рассчитывается по формуле (1.5).

(1.5)

Ассоциативное правило является *частым*, если набор XY является *частым*, то есть и является *надежным*, если , где *minconf* – заданный пользователем минимальный порог доверия [7].

Исходя из определений поддержки и достоверности ассоциативных правил, можно сказать, что задача поиска ассоциативных правил состоит из двух этапов:

1. поиск всех частых наборов с их значениями поддержки;
2. поиск всех частых ассоциативных правил, удовлетворяющих критерию надежности.
   * 1. Подходы к анализу фискальных данных

Поскольку первый этап решения задачи поиска ассоциативных правил заключается в поиске всех частых наборов с их значениями поддержки, необходимо рассмотреть существующие алгоритмы поиска частых наборов. При рассмотрении алгоритмов необходимо оценить их вычислительную сложность, т.е. функцию зависимости объёма работы алгоритма, от размера входных данных. Дополнительно, поскольку алгоритм будет непосредственно работать с базой данных, необходимо также учитывать количество операций ввода/вывода, выполняемое алгоритмом в процессе его работы.

Далее рассмотрены четыре алгоритма поиска частых наборов, а также приведены результаты их работы на примере базы данных, показанной на рисунке 1.2.1.

### Алгоритм BRUTEFORCE

Наиболее простым и примитивным алгоритмом поиска частых наборов является алгоритм BRUTEFORCE, который перечисляет все возможные наборы и для каждого такого подмножества определяет его поддержку во входном наборе данных.

Данный алгоритм состоит из двух основных этапов:

1. Генерация кандидатов - генерирует все подмножества набора элементов, которые называются кандидатами, каждый из которых потенциально является частым набором.
2. Вычисление поддержки - вычисляется поддержка каждого кандидата и определяется, является ли он частым.

На рисунке 1.2.2 показана решетка наборов множества элементов . Существует возможных наборов, включая пустой набор. Также на данном рисунке полужирным шрифтом выделено префиксное дерево, образованное частыми наборами при минимальной поддержке minsup = 3.

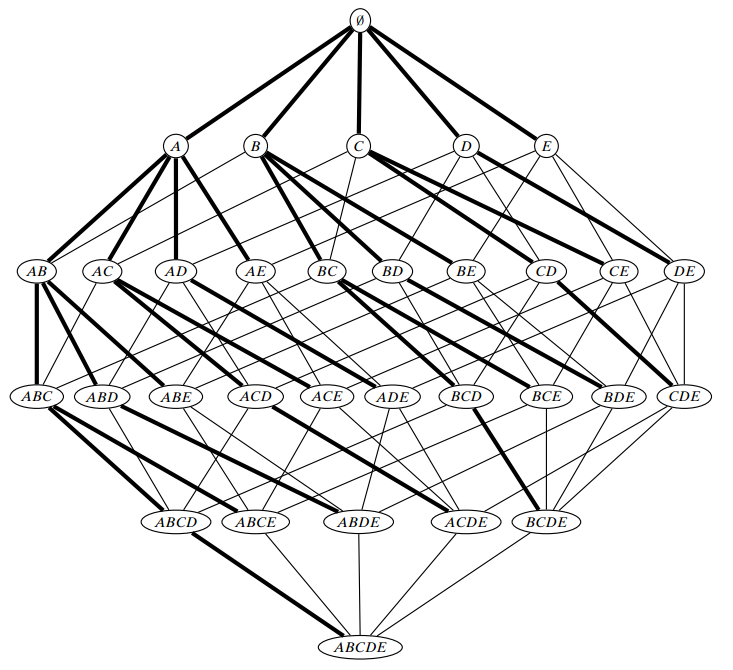


Рисунок 1.2.2 – Пример построения решетки наборов алгоритмом BRUTEFORCE

Вычислительная сложность данного алгоритма складывается из сложности вычисления поддержки для каждого кандидата, которая в худшем случае . Учитывая, что количество кандидатов , сложность алгоритма BRUTEFORCE будет .

Поскольку мы выполняем одно полное сканирование базы данных для вычисления поддержки каждого кандидата, сложность ввода-вывода BRUTEFORCE - это проходов по базе данных. Таким образом, подход BRUTEFORCE является вычислительно невыполнимым даже для небольших наборов данных, которые на практике могут быть очень большими (например, супермаркет, в котором продаются тысячи продуктов). Этот подход также нецелесообразен и с точки зрения ввода-вывода.

### Алгоритм Apriori

Данный алгоритм использует свойство анти-монотонности поддержки, смысл которого заключается в том, что поддержка любого набора элементов не может превышать минимальной поддержки любого из его подмножеств. Можно сформулировать данное утверждение иначе: с ростом размера набора его поддержка уменьшается, либо остается такой же. Это означает, что любой k-элементный набор будет являться частым, тогда и только тогда, когда все его подмножества будут частыми наборми. Также алгоритм Apriori использует поуровневый подход, т.е. на каждой итерации он вычисляет поддержку наборов одного уровня, и если она оказывается меньше минимальной, то надмножества такого набора рассматриваться не будут.

Пример работы данного алгоритма приведен на рисунке 1.2.3. Каждый узел указывает на набор элементов и свою поддержку.

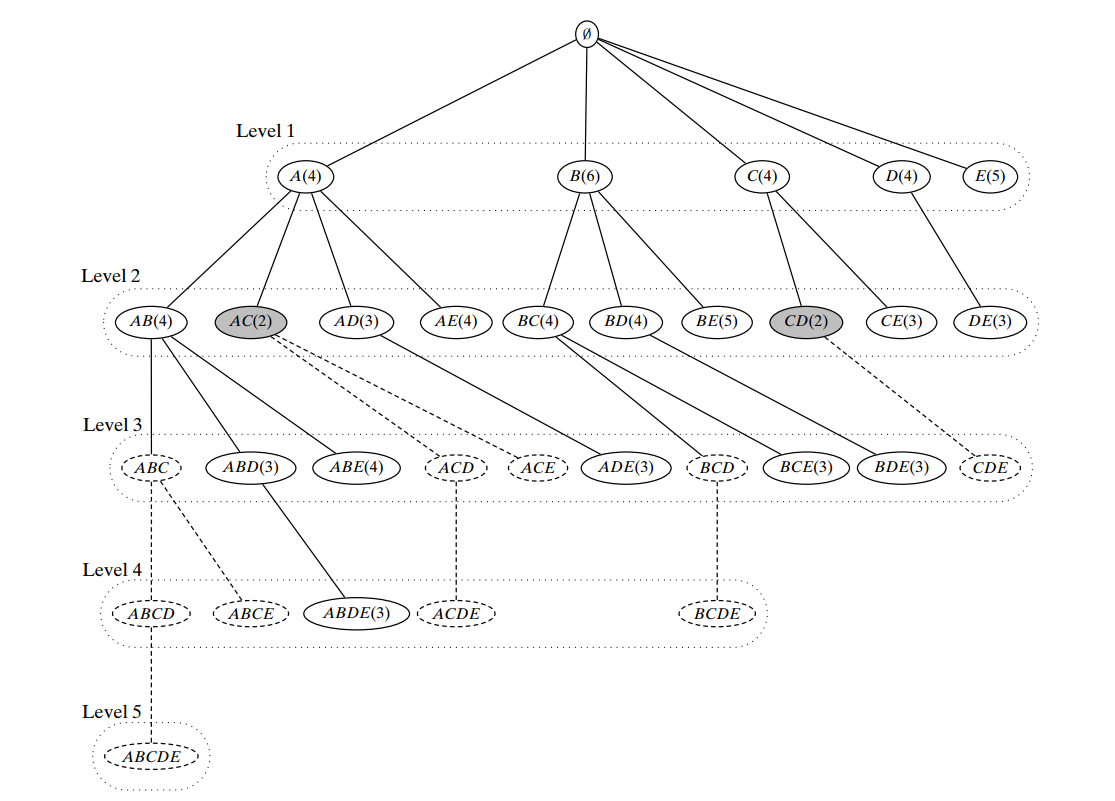


Рисунок 1.2.3 – Пример построения префиксного дерева алгоритмом Apriori

В наихудшем случае вычислительная сложность алгоритма Apriori по-прежнему равна , так как все элементы набора могут быть частыми. Но на практике из-за сокращения пространства поиска вычислительные затраты становятся намного ниже. С точки зрения затрат на ввод / вывод Apriori требует проходов по базе данных, в отличие от в алгоритме BRUTEFORCE.

### Алгоритм Eclat

Еще одним алгоритмом поиска частых наборов является алгоритм Eclat. Для своей работы он использует вертикальное представление базы данных (см. рисунок 1.2.4 – б).

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | t | X | | 1 | ABDE | | 2 | BCE | | 3 | ABDE | | 4 | ABCE | | 5 | ABCDE | | 6 | BCD | | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | TID-set | A | B | C | D | E | | t(x) | 1  3  4  5 | 1  2  3  4  5  6 | 2  4  5  6 | 1  3  5  6 | 1  2  3  4  5 | |
| (а) | (б) |

Рисунок 1.2.4 – Представление базы данных транзакций

а – БД транзакций, б – вертикальная БД

Здесь - это набор идентификаторов транзакций, которые содержат один элемент .

Алгоритм Eclat использует наборы идентификаторов транзакций (TID-sets) непосредственно для вычисления поддержки. Основная идея заключается в том, что поддержка набора кандидатов может быть вычислена путем пересечения TID-sets подходящих выбранных подмножеств.

В общем случае при *t (X)* и *t (Y)* для любых двух частых наборов *X* и *Y* мы имеем . Поддержка кандидата *XY* - это просто мощность *t (XY)*, т. е. Eclat пересекает TID-наборы только в том случае, если частые наборы имеют общий префикс. Группа наборов элементов, имеющих один и тот же префикс, называется *классом эквивалентности префикса.* Пример построения набора пересекающихся транзакций алгоритмом Eclat приведен на рисунке 1.2.5.

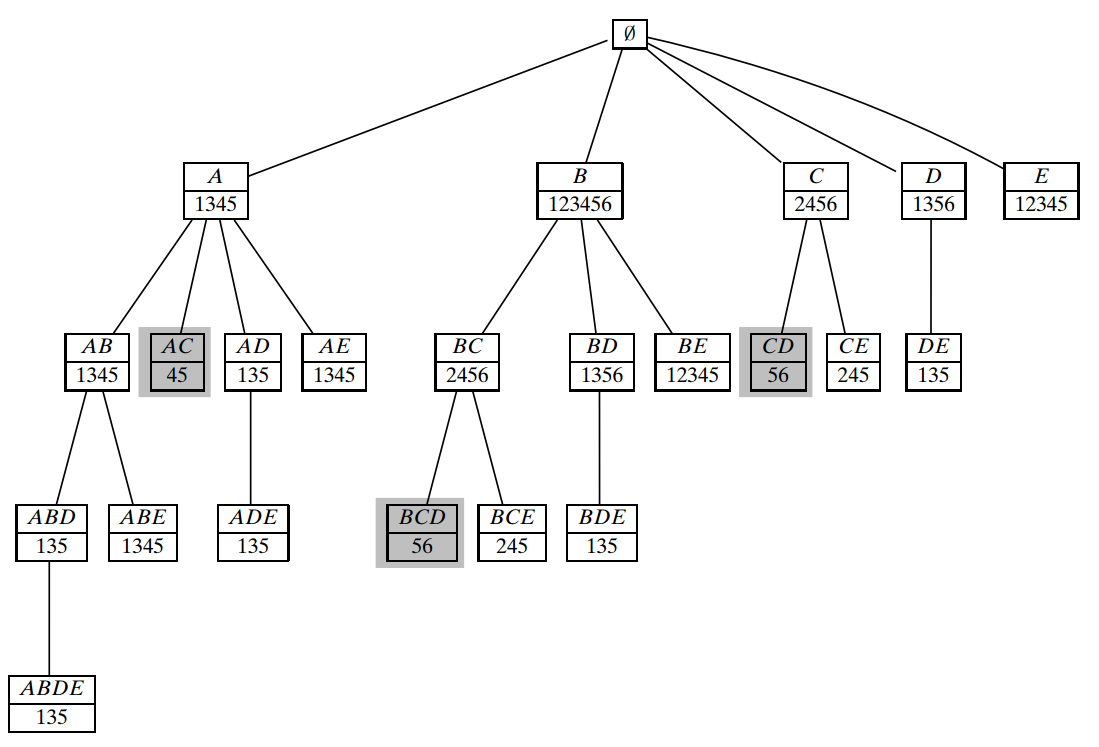


Рисунок 1.2.5 – Пример построения набора пересекающихся транзакций алгоритмом Eclat

В худшем случае сложность вычислений Eclat равна , так как может существовать частых наборов и пересечение двух TID-наборов занимает не более Сложность ввода-вывода Eclat описать сложнее, поскольку она зависит от размера промежуточных веток. Если обозначить средний размер TID-набора как *t*, то исходный размер базы данных будет а общий размер всех промежуточных веток - это Таким образом, в худшем случае Eclat требует проходов по базе данных.

### Алгоритм Frequent Pattern-Growth Strategy (FPG)

Одним из наиболее эффективных алгоритмов поиска ассоциативных правил является алгоритм Frequent Pattern-Growth (алгоритм FPG). В основе этого метода лежит предобработка базы данных транзакций, при которой эта база данных преобразуется в компактную древовидную структуру, называемую Frequent-Pattern Tree – префиксное дерево частых предметных наборов (FP-дерево).

К основным преимуществам данного метода можно отнести:

* Сжатие БД транзакций в компактную структуру, что обеспечивает очень эффективное и полное извлечение частых предметных наборов;
* При построении FP-дерева используется технология разделения и захвата (англ.: divide and conquer), которая позволяет выполнить декомпозицию одной сложной задачи на множество более простых;
* Данный алгоритм позволяет избежать затратной процедуры генерации кандидатов (наборов элементов, каждый из которых потенциально является частым набором), характерной для таких алгоритмов, как Apriori и Elcat.

Алгоритм FPGrowth состоит из двух этапов: построения FP-дерева и извлечение частых наборов из построенного дерева [7].

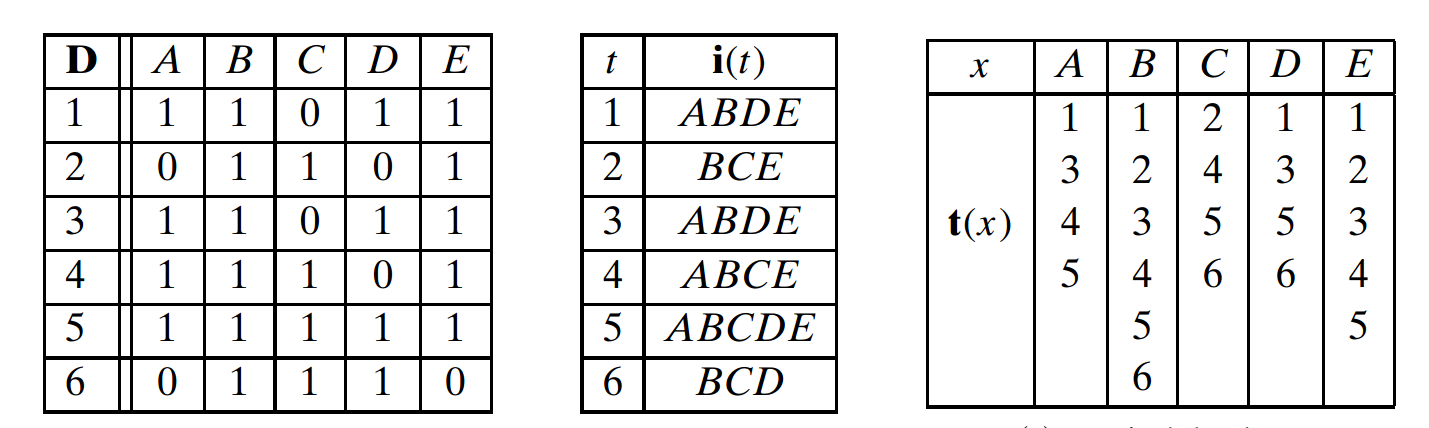
#### 1.2.6.1 Построение FP-дерева

Как уже было сказано, алгоритм FPGrowth индексирует базу данных для быстрого вычисления поддержки с использованием префиксного дерева. Каждый узел в дереве является одиночным элементом, а каждый дочерний узел является отличным от родительского. Каждый узел также хранит информацию о поддержке для набора элементов, содержащих элементы на пути от корня к этому узлу.

FP-дерево строится следующим образом. Первоначально дерево содержит в качестве корня нулевой элемент ∅. Далее, для каждого набора (элементов транзакции) , где мы вставляем данный набор элементов в FP-дерево, увеличивая счетчик всех узлов вдоль пути, который представляет . Если имеет префикс с элементами ранее вставленной транзакции, тогда будет следовать одному и тому же пути до общего префикса. Для остальных элементов в , новые узлы создаются под общим префиксом, при этом счетчики инициализируются с 1. Построение FP-дерева завершается после вставки всех транзакций.

FP-дерево можно рассматривать как сжатое префиксное представление базы данных. Для того, чтобы дерево было как можно более компактным, необходимо, чтобы наиболее часто используемые элементы находились в верхней части дерева. Поэтому FPGrowth переупорядочивает элементы в порядке убывания поддержки, то есть из исходной базы данных сначала вычисляется поддержка всех отдельных элементов . Затем редкие элементы отбрасываются, а оставшиеся элементы сортируются в порядке уменьшения поддержки. Наконец, каждый набор вставляется в FP-дерево после переупорядочения X путем уменьшения поддержки элемента.

Рассмотрим пример БД на рисунке 1.2.6.

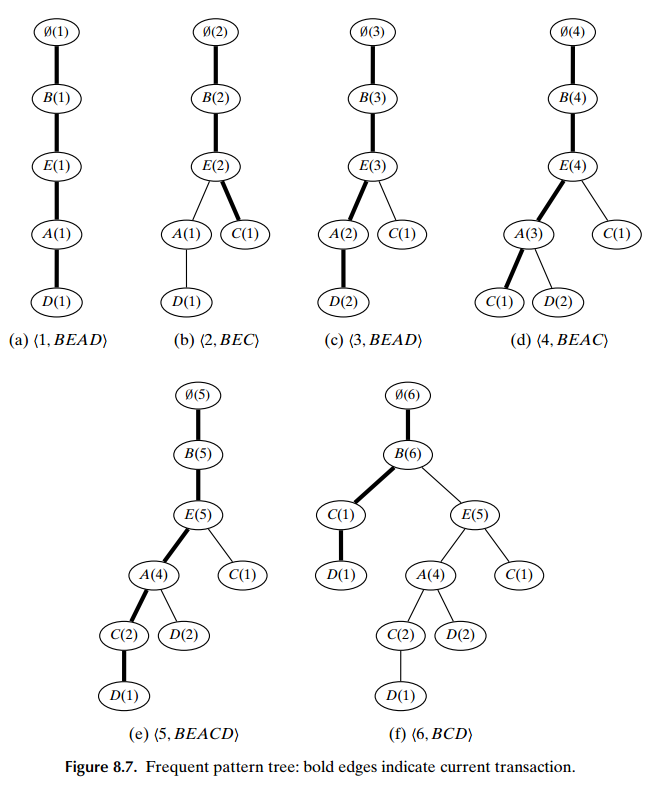


|  |  |  |
| --- | --- | --- |
| (а) Бинарная БД | (б) БД транзакций | (в) Вертикальная БД |

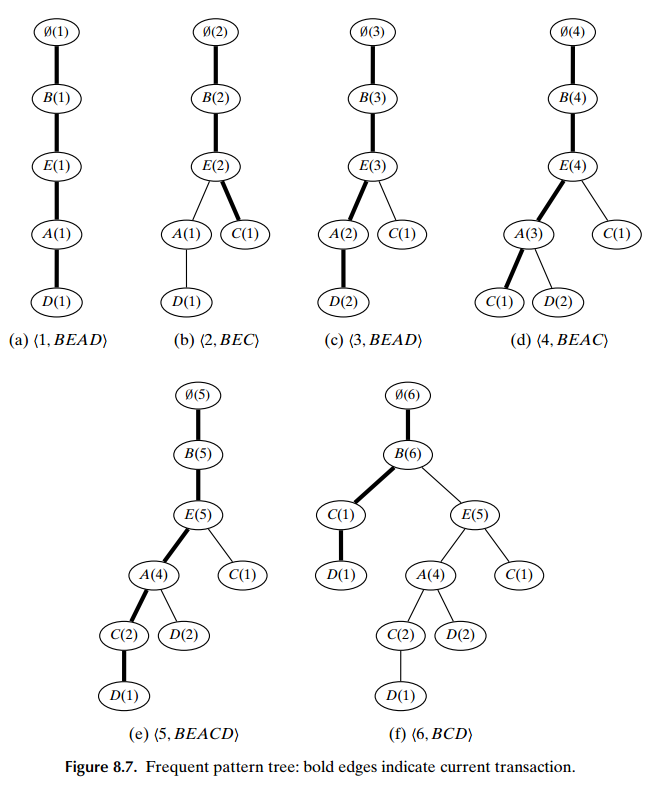
Рисунок 1.2.6 – Пример БД

Для построения дерева, каждая транзакция поочерёдно упорядочивается и добавляется в FP-дерево. Для приведенного примера БД, упорядоченный набор элементов – имеет вид {B (6), E (5), A (4), C (4), D (4)}. Например, становится . На рисунке 1.2.7 показана поэтапная конструкция FP-дерева. Окончательный вид FP-дерева для базы данных показано на рисунке 1.2.7 (е).

После того как FP-дерево было построено, оно служит в качестве индекса вместо исходной базы данных. Все частые элементы могут быть извлечены из дерева напрямую с помощью метода FPGROWTH. Метод принимает на вход FP-дерево *R*, построенное из входной базы данных D, и префикса текущего набора *P*, который изначально пуст.



|  |  |  |  |
| --- | --- | --- | --- |
| (а) | (б) | (в) | (г) |



|  |  |  |
| --- | --- | --- |
|  | (д) | (е) |

Рисунок 1.2.7 – Построение FP дерева (жирные ребра показывают текущую транзакцию)

#### 1.2.6.2 Извлечение частых наборов из FP-дерева

Процедура извлечения частых наборов из FP-дерева состоит из следующих этапов:

1. Выбор элемента из БД;
2. Построение условного FP-дерева для выбранного элемента;
3. Получение частых наборов путем обхода с вершины каждой ветви условного FP-дерева;

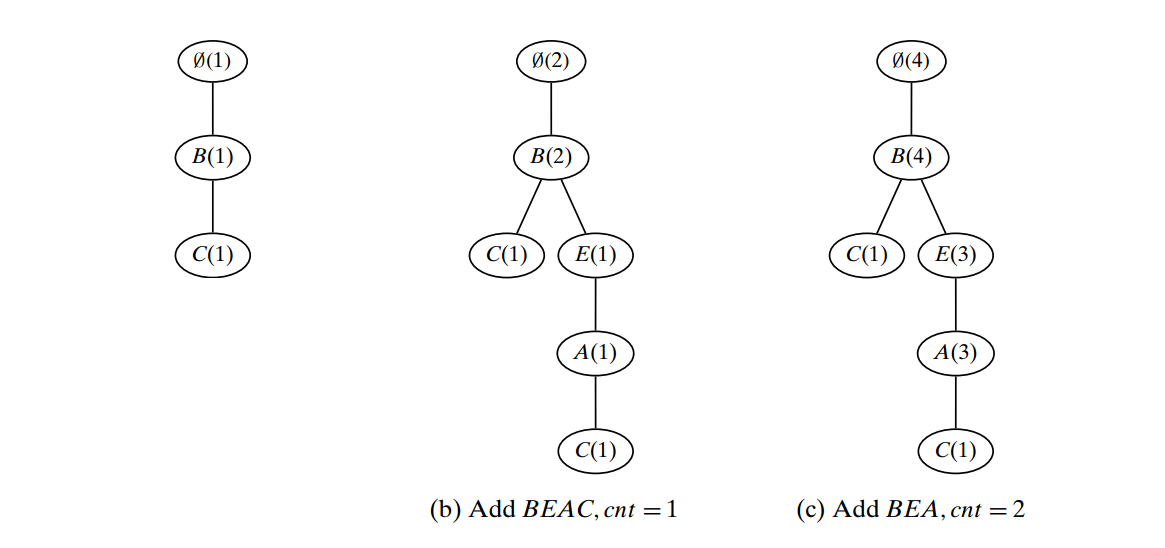
Данная процедура итеративно производится для каждого элемента, встречающегося в БД, после чего, полученные на каждой итерации частые наборы объединяются в общий список.

Построение условного FP-дерева для выбранного элемента производится следующим образом: для выбранного предмета сохраняются все пути, ведущие от каждого узла с данным предметом к корню дерева. Такой набор путей называется условным базисом предмета. Каждый путь в базисе состоит из двух частей – префикса и суффикса. Префикс – это последовательность узлов, через которые нужно пройти чтобы достичь узла, связанного с предметом, а суффиксом является сам узел. Далее из каждого элемента условного базиса удаляется суффикс, после чего условный базис будет состоять только из префиксов. Далее вычисляется сколько раз каждый предмет появляется в префиксах путей, полученных на предыдущем шаге, и все элементы условного базиса упорядочиваются в порядке убывания этих значений, откуда получается новый набор транзакций, на основании которых строится новое FP-дерево, называемое условным.

Далее в условном FP-дереве находятся все предметы (узлы), для которых поддержка (количество появлений в дереве) соответствует заданному уровню минимальной поддержки. Если предмет встречается несколько раз, то его индексы, т.е. частоты появлений в условном базисе, суммируются. После этого, в конец каждой ветви дерева возвращается удаленный ранее суффикс, и начиная с верхушки дерева, сохраняются пути, которые ведут к каждому из возвращенных узлов.

Проведем процедуру извлечения частых наборов на построенном ранее FP-дереве R (см. рисунок 1.2.7-е). Пусть порог поддержки minsup = 3. Множество частых элементов i встречающихся в дереве R: B (6), E (5), A (4), C (4) и D (4).

На рисунке 1.2.8 изображен процесс построения условного FP-дерева для предмета D, где показано, что из корня есть три пути к узлу, помеченному D, а именно: BCD с поддержкой 1; BEACD с поддержкой 1; BEAD с поддержкой 2. Эти три пути, исключая последний элемент i = D, вставляются в новое дерево RD, в котором вычисляются частоты появлений каждого элемента. Среди элементов дерева заданному уровню минимальной поддержки не удовлетворяет элемент С, встречающийся в дереве 2 раза, поэтому он исключается, после чего полученное FP-дерево будет представлять собой единственный путь B (4) -E (3) - А (3). Таким образом, мы перечисляем все подмножества этого пути и приписываем их D, чтобы получить часто используемые элементы BD (4), ED (3), AD (3), BED (3), BAD (3), EAD (3), и BEAD (3).



|  |  |  |
| --- | --- | --- |
| (а) BC (1) | (б) BEAC (1) | (в) BEA (2) |

Рисунок 1.2.8 – Построение условного FP-дерева для предмета D

Аналогичным образом обрабатываются оставшиеся предметы. Все проецируемые деревьям для C, A и E являются однопутными, что позволяет найти частые наборы {CB (4), CE (3), CBE (3)}, {AE (4), AB (4), AEB (4)} и {EB (5)} соответственно. Этот процесс показан на рисунке 1.2.9.

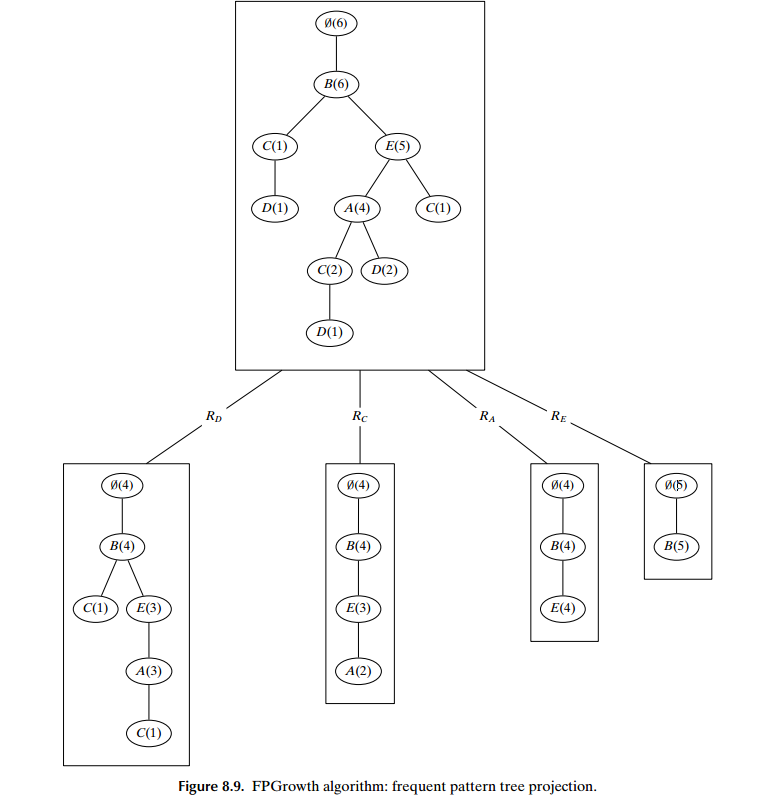


Рисунок 1.2.9 – Построение условных FP-деревьев для каждого предмета

На основе частых наборов, полученных в результате работы алгоритма можно выполнить генерацию ассоциативных правил.

### Поиск оптимального подхода к поиску ассоциативных правил

Поскольку в рамках данной работы решается задача динамического построения модели данных, предназначенной для быстрого поиска ассоциативных правил [6], то ни один из рассмотренных подходов в своем исходном виде не подходит для решения поставленной задачи. Это объясняется тем, что они предназначены для обработки, имеющейся в наличии, полной базы данных с транзакциями, а поставленная задача подразумевает возможность работы с изменяющейся во времени базой данных.

Из существующих алгоритмов для поставленной задачи наиболее подходящим является Frequent Pattern-Growth. Это объясняется следующими его свойствами. Во-первых, он логически разделен на два этапа:

* Построение FP-дерева;
* Извлечение частых наборов из FP-дерева.

Это означает, что данные этапы могут выполнятся независимо друг от друга в различные моменты времени.

Еще одной особенностью данного алгоритма является то, что на первом этапе он преобразует имеющуюся базу данных в компактную и удобную для обработки структуру – префиксному дереву. Таким образом можно постепенно собирать данное префиксное дерево и выполнять второй этап алгоритма именно в тот момент, когда в этом появляется потребность. Кроме того, благодаря своей компактной структуре префиксное дерево можно кешировать, тем самым повышая скорость обработки запросов.

Также необходимо учитывать, что для того, чтобы построенное дерево имело как можно более компактную и сбалансированную структуру, необходимо добавлять в него транзакции в упорядоченном виде (в порядке убывания поддержки). Однако вышеописанный подход не позволяет полностью выполнять это условие, поскольку с течением времени поддержка отдельных элементов, и соответственно схема упорядочивания элементов в транзакциях, может меняться. Частично компенсировать данный недостаток, можно осуществляя накопление транзакций и обновляя дерево не при регистрации каждой новой транзакции, а при возникновении специального события, например, по истечению заданного временного интервала. Это позволит избежать несбалансированного построения дерева в ситуациях, когда на малом промежутке времени рост поддержки отдельных элементов является несистемным.

## 1.3 Выводы

В ходе выполнения анализа предметной области, были определены понятие и структура фискальных данных, из чего был сделан вывод, что фискальные данные являются потенциальным источником ценной информации, а ОФД, имея доступ к этим данным, может производить их анализ, с целью получения ценной информации. Для извлечения ценной информации из большого количества данных, необходимо использовать инструменты интеллектуального анализа данных (Data Mining).

Дополнительно были рассмотрены несколько существующих решений, применяющих методы интеллектуального анализа данных в приложении к аналитике фискальных данных.

Таким образом было установлено, что разработка специализированных инструментов анализа фискальных данных является актуальной и коммерчески обоснованной задачей.

На основании проведенного анализа предметной области была сформулирована бизнес-задача аналитики фискальных данных, заключающаяся в анализе рыночной корзины, на основании накопленных за определенный период фискальных данных. С точки зрения интеллектуального анализа данных, поставленная задача является задачей поиска ассоциативных правил, для которой была приведена математическая формулировка.

Далее было установлено, что задача поиска ассоциативных правил состоит из двух этапов:

1. поиск всех частых наборов с их значениями поддержки;
2. поиск всех частых ассоциативных правил, удовлетворяющих критерию надежности.

Поскольку первый этап решения задачи поиска ассоциативных правил заключается в поиске всех частых наборов с их значениями поддержки, были рассмотрены существующие алгоритмы поиска частых наборов.

При рассмотрении алгоритмов были оценены их вычислительная сложность, а также количество операций ввода/вывода, выполняемое алгоритмом в процессе его работы.

Из рассмотренных алгоритмов для поставленной задачи наиболее подходящим является Frequent Pattern-Growth. Это объясняется тем, что он логически разделен на два этапа:

* Построение FP-дерева;
* Извлечение частых наборов из FP-дерева.

Это означает, что данные этапы могут выполнятся независимо друг от друга в различные моменты времени.

# Проектно-конструкторская часть

## Функциональные требования

Функциональность, которую необходимо реализовать в системе можно графически представить в виде Use Case диаграммы. Use Case диаграмма проектируемой системы представлена на рисунке 2.1.

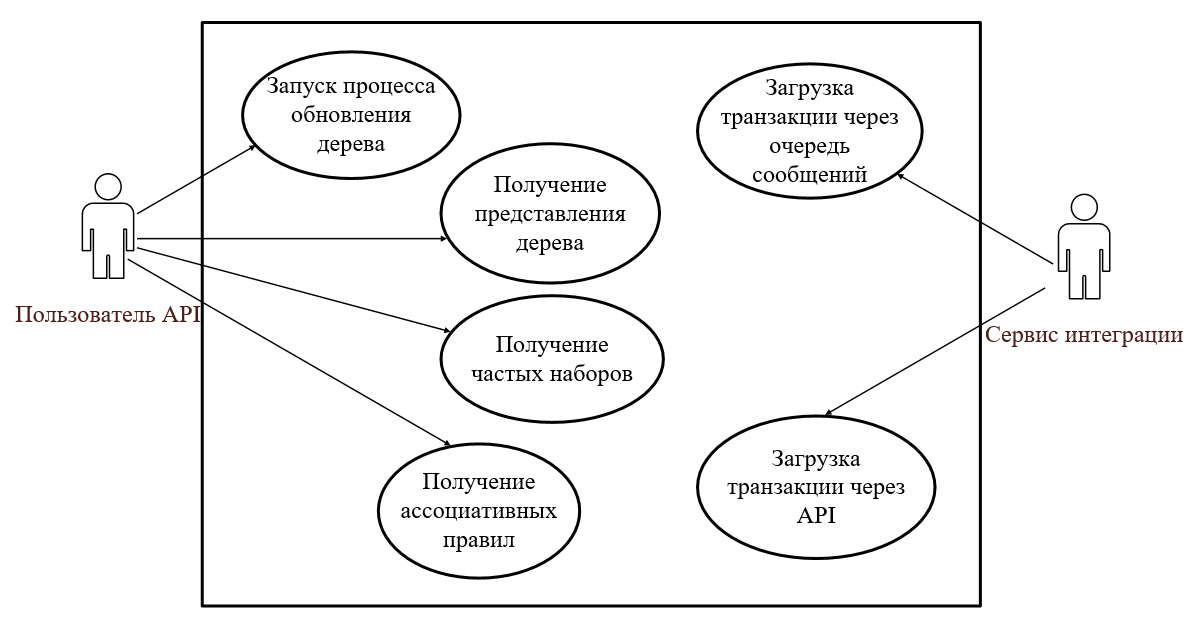


Рисунок 2.1 – Use Case диаграмма системы

Проектируемая система может взаимодействовать с двумя действующими лицами: пользователем системы и сервисом интеграции с ОФД. Пользователь системы имеет возможность взаимодействовать через API с аналитическим модулем системы:

* Получать представление префиксного дерева;
* Получать частые наборы из текущей версии дерева;
* Получать ассоциативные правила из текущей версии дерева;
* Вручную запускать процесс обновления дерева.

Сервис интеграции с ОФД имеет возможность загружать в систему новые транзакции двумя способами: через API и через сервис очередей.

## Нефункциональные требования

Для того, чтобы упростить процесс разработки, а также обеспечить возможность совершенствования системы в будущем, необходимо при разработке системы обеспечить выполнение следующих требований:

* Расширяемость;
* Масштабируемость;
* Кроссплатформенность.

Расширяемость – это возможность сравнительно легкого добавления в систему дополнительной функциональности. Расширяемость может быть достигнута за счет гибкой архитектуры.

Масштабируемость – это возможность увеличивать вычислительную мощность системы, за счет увеличения вычислительных ресурсов. Масштабируемость системы может быть достигнута за счет оптимального использования вычислительных ресурсов, с которыми будет работать система.

Кроссплатформенность означает что система может работать на различных операционных системах, таких как Windows, Mac OS X, Linux и др. Кроссплатформенность позволяет разработчику создавать приложения без привязки к конкретной операционной системе. Возможность работы приложения на различных операционных системах зависит от выбора технологии разработки, т.к. одни технологии позволяют делать приложения кроссплатформенными, в то время, как другие позволяют разрабатывать приложения только для конкретных ОС.

## Архитектура системы

Основным этапом разработки системы является проектирование архитектуры. Важно, чтобы архитектура системы была расширяемой, т.е. позволяла без затруднений добавлять в систему дополнительную функциональность, а также масштабируемой, т.е. позволяла увеличивать вычислительную мощность системы, за счет увеличения вычислительных ресурсов. Поэтому, система должна быть представлена как совокупность отдельных модулей, а не как монолит. Учитывая вышесказанное, основной задачей проектирования архитектуры системы будет являться определение модулей системы и их ответственности.

Поскольку проектируемая система предназначена для накопления и анализа фискальных данных, то в ней должны присутствовать соответствующие модули. Также, учитывая, что данные в систему будут поступать извне, то появляется необходимость в модуле предварительной обработки данных, задачей которого будет фильтрация поступающих данных и их преобразование к внутреннему формату системы. Кроме того, необходимо учесть, что источник данных может передавать большие объемы данных за малые промежутки времени, и данный факт не должен отрицательно сказываться на работоспособности системы.

В соответствии с поставленными требованиями была описана структура системы, общий вид которой представлен на рисунке 2.2.

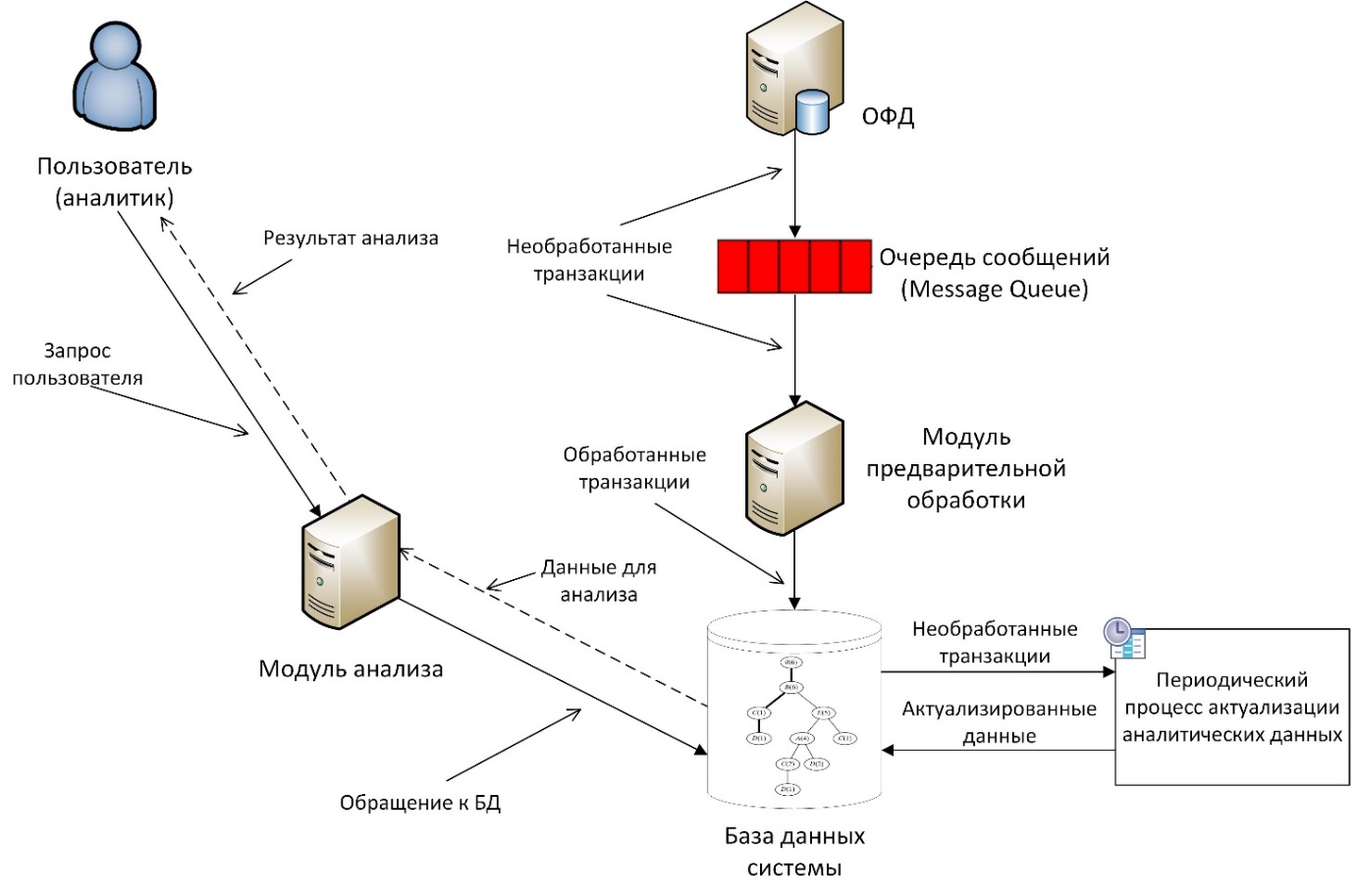


Рисунок 2.2 - Общий вид системы

На данной схеме приведены вышеописанные модули, а также их связи друг с другом и с внешними, по отношению к системе, сущностями. Таким образом источник данных (транзакций), обозначенный в схеме как «ОФД», связан с модулем предварительной обработки не на прямую, а посредством очереди сообщений. Данный подход позволит во-первых – оградить модуль предварительной обработки от перегрузки, во-вторых – избежать потери данных, если модуль обработки по какой-либо причине будет недоступен, например, во время обновления системы.

При получении новой транзакции, модуль предварительной обработки будет производить ее обработку следующим образом: извлекать из транзакции требуемые данные, т.е. список элементов (товаров) участвующих в транзакции, преобразовать эти данные к внутреннему формату системы, после чего записывать эту информацию в БД, с сохранением времени записи. На основе сохраненных транзакций будет строиться специальная структура данных, которая позволит эффективно извлекать аналитическую информацию с которой далее будет работать модуль анализа. Однако с течением времени данная структура становится неактуальной, т.к. в систему постоянно будут поступать новые транзакции и соответственно созданную ранее структуру данных необходимо актуализировать. Для этого во время работы системы будет запущена специальная служба, которая через определенные промежутки времени будет извлекать из БД транзакции, фильтруя их по временному признаку и дополнять текущую структуру новыми данными. Таким образом модуль анализа будет иметь возможность получать последнюю актуальную версию данных из БД и кешировать ее, во избежание задержки на запрос к БД во время пользовательского запроса. Обоснование использования данной архитектуры также описано в исследовании [8].

## Проектирование базы данных системы

Одним из основополагающих компонентов системы является база данных, хранящая данные, необходимые для функционирования системы. Проектируемая БД должна удовлетворять следующим требованиям:

* Возможность хранения в компактном виде регистрируемых в системе транзакций;
* Возможность фильтрации сохраненных транзакций про временному признаку;
* Возможность хранения специальной иерархической структуры данных, для извлечения аналитической информации;
* Возможность обновления вышеописанной структуры и хранения информации об этих обновлениях;

### Концептуальное проектирование БД

Первым этапом проектирования базы данных является создание концептуальной модели данных, которая базируется на анализе свойств и природы объектов предметной области и информационных потребностей будущих пользователей разрабатываемой системы.

Исходя из поставленных требований, в системе можно выделить следующие сущности:

* Транзакция (Transaction) – объект, хранящий информацию наборе продуктов, указанных в одном чеке;
* Продукт (Product) – класс, характеризующий продукты, встречающиеся в транзакциях, которые относятся к одной группе;
* Узел (Node) – Элемент иерархической структуры.

На основании поставленных требований была спроектирована логическая модель базы данных, приведенная на рисунке 2.3.

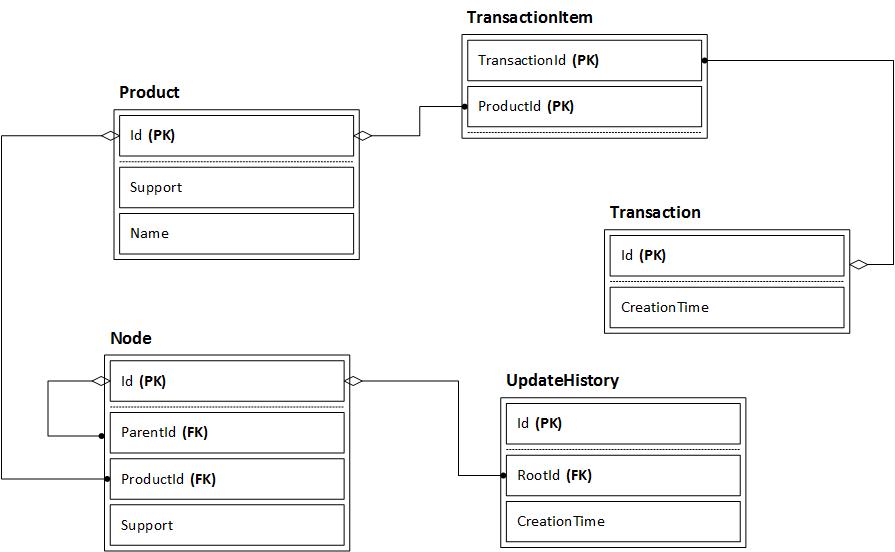


Рисунок 2.3 - Логическая модель базы данных

В ходе проектирования приведенной модели были выделены две дополнительных сущности:

* Элемент транзакции (Transaction Item) – является развязкой отношения «многие ко многим» между сущностями «Product» и «Transaction»;
* История обновлений (Update History) – содержит информацию об изменениях, вносимых в иерархическую структуру.
  + 1. Физическая модель данных

Физическое проектирование — создание схемы базы данных для конкретной СУБД. Отношения, разработанные на стадии формирования логической модели данных, преобразуются в таблицы, атрибуты становятся столбцами таблиц, для ключевых атрибутов создаются уникальные индексы, домены преображаются в типы данных, принятые в конкретной СУБД.

Для описания атрибутов сущностей проектируемой базы данных были использованы следующие типы СУБД PostgreSQL:

* uuid – глобальный уникальный идентификатор;
* serial – автоинкрементный целочисленный тип;
* integer – 4х-байтные целочисленные данные;
* character varying (varchar) – символьные данные переменной длины;
* timestamp with time zone – Дата и время с указанием часового пояса.

Далее, в таблицах 2.1 – 2.5 приведены полные описания атрибутов каждой сущности.

Таблица 2.1– Сущность «Транзакция» (Transaction)

|  |  |  |
| --- | --- | --- |
| **Атрибут** | **Тип** | **Описание** |
| Id | uuid | Уникальный идентификатор |
| CreationTime | timestamp with time zone | Дата и время сохранения |

Таблица 2.2– Сущность «Продукт» (Product)

|  |  |  |
| --- | --- | --- |
| **Атрибут** | **Тип** | **Описание** |
| Id | uuid | Уникальный идентификатор |
| Support | integer | Поддержка (общее количество появлений продукта среди всех зарегистрированных транзакций) |
| Name | varchar | Наименование |

Таблица 2.3– Сущность «Элемент транзакции» (Transaction Item)

|  |  |  |
| --- | --- | --- |
| **Атрибут** | **Тип** | **Описание** |
| TransactionId | uuid | Внешний ключ к «Transaction» и часть уникального идентификатора |
| ProductId | uuid | Внешний ключ к «Product» и часть уникального идентификатора |

Таблица 2.4 – Сущность «Узел» (Node)

|  |  |  |
| --- | --- | --- |
| **Атрибут** | **Тип** | **Описание** |
| Id | uuid | Уникальный идентификатор |
| ParentId | uuid | Идентификатор родительского узла (Внешний ключ) |
| ProductId | uuid | Внешний ключ к «Product» |
| Support | integer | Поддержка продукта в узле дерева |

Таблица 2.5 – Сущность «История обновлений» (Update History)

|  |  |  |
| --- | --- | --- |
| **Атрибут** | **Тип** | **Описание** |
| Id | serial | Уникальный идентификатор |
| RootId | uuid | Идентификатор корня обновленного дерева (Внешний ключ к «Node») |
| CreationTime | timestamp with time zone | Дата и время обновления дерева |

## Проектирование решения

Проектируемое решение будет состоять из следующего набора сборок:

* SenseMining.Entities - Содержит модели-сущности для работы ORM;
* SenseMining.Database - Содержит контекст данных для работы ORM;
* SenseMining.Domain - Содержит бизнес-логику системы;
* SenseMining.Worker - Исполняемое приложение. Выполняет автоматическое обновление префиксного дерева с заданным интервалом;
* SenseMining.API – ASP .NET Core приложение, содержит HTTP API для работы с системой;
* SenseMining.Listener - Исполняемое приложение. Выполняет прослушивание и обработку сообщений из очереди;
* SenseMining.Importer.GroceryMarket – Приложение для импорта данных в систему для ее отладки;
* SenseMining.Domain.MessageContracts – Сборка с контрактами MQ-сообщений между SenseMining.Importer.GroceryMarket и SenseMining.Listener.

Схематическое представление зависимостей данных сборок проекта системы приведено на рисунке 2.4.

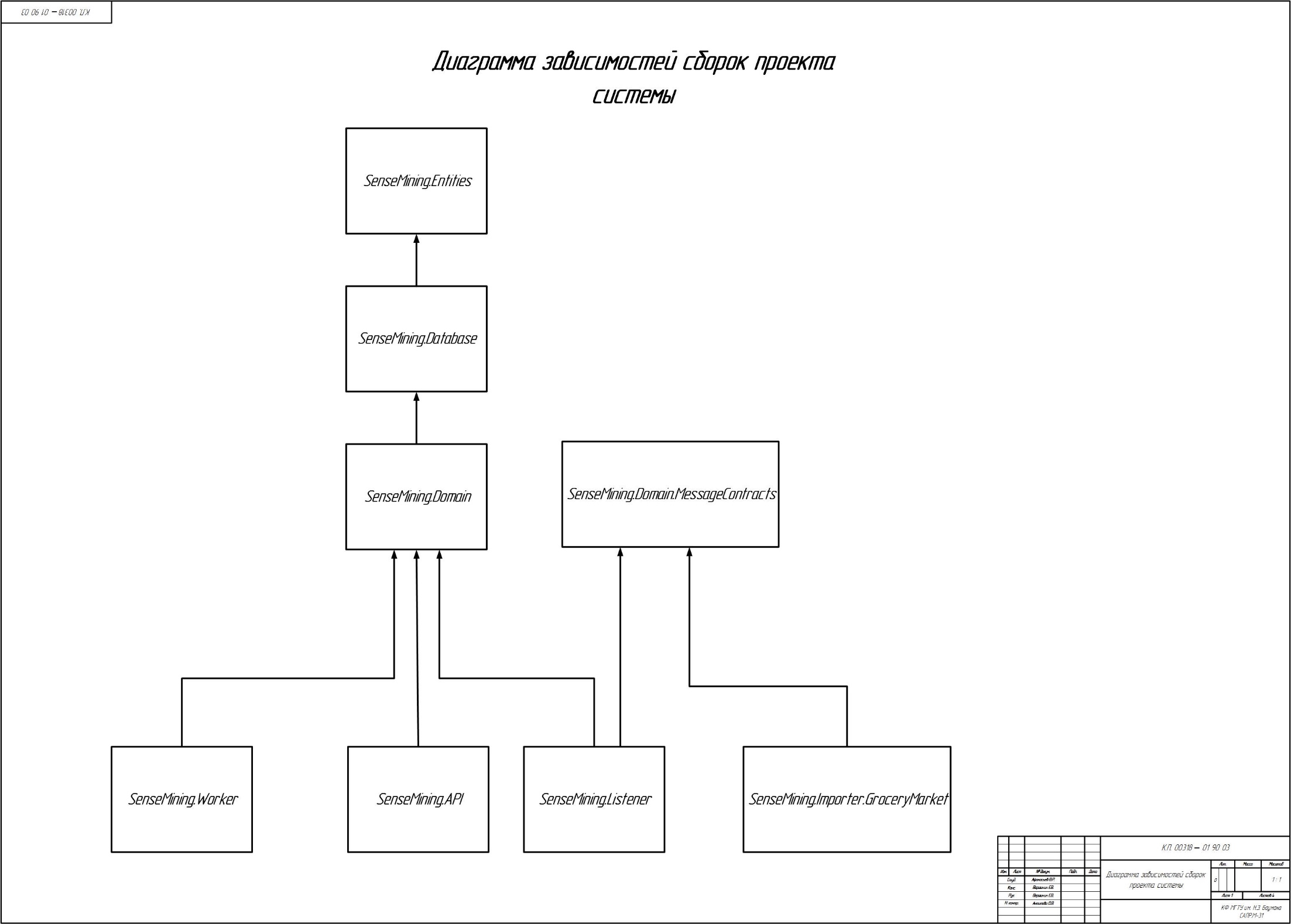


Рисунок 2.4 - Диаграмма зависимостей сборок проекта системы

# Проектно-технологическая часть

## Платформа для разработки системы

В качестве платформы для разработки проектируемой системы была выбрана платформа .NET Core. Данная платформа обладает всеми необходимыми инструментами, для построения гибкой, расширяемой и масштабируемой системы.

.NET Core — это универсальная платформа разработки, которая поддерживается корпорацией Майкрософт и сообществом .NET на сайте [GitHub](https://github.com/dotnet/core). Она является кроссплатформенной, поддерживает Windows, Mac OS и Linux. Перечисленные ниже особенности наиболее полно определяют платформу .NET Core:

* Гибкая разработка: может включаться в приложение или устанавливаться параллельно на уровне пользователя или компьютера;
* Кроссплатформенность: работает в Windows, Mac OS и Linux; может переноситься в другие операционные системы. Спектр [поддерживаемых операционных систем (ОС)](https://github.com/dotnet/core/blob/master/roadmap.md), ЦП и приложений будет со временем расширяться благодаря усилиям корпорации Майкрософт, других компаний и отдельных лиц;
* Программы командной строки: любые сценарии использования продукта можно реализовать посредством командной строки;
* Совместимость: платформа .NET Core совместима с .NET Framework, Xamarin и Mono благодаря [.NET Standard](https://docs.microsoft.com/ru-ru/dotnet/standard/net-standard);
* Открытый исходный код: платформа .NET Core имеет открытый исходный код и распространяется по лицензиям MIT и Apache 2. Документация распространяется по лицензии [CC-BY](https://creativecommons.org/licenses/by/4.0/). .NET Core является проектом [.NET Foundation](https://dotnetfoundation.org/);
* Поддержка корпорации Майкрософт: платформа .NET Core поддерживается корпорацией Майкрософт согласно правилам [жизненного цикла поддержки .NET Core](https://www.microsoft.com/net/core/support/);

.NET Core состоит из перечисленных ниже компонентов:

* [Среда выполнения .NET](https://github.com/dotnet/coreclr) предоставляет систему типов, функцию загрузки сборок, сборщик мусора, взаимодействие с машинным кодом и другие основные службы;
* Набор [библиотек платформы](https://github.com/dotnet/corefx) предоставляет примитивные типы данных, типы компоновки приложений и базовые служебные программы;
* Набор [средств SDK](https://github.com/dotnet/cli) и [компиляторы языков](https://github.com/dotnet/roslyn) обеспечивают базовые возможности разработки, доступные в [пакете SDK для .NET Core](https://docs.microsoft.com/ru-ru/dotnet/core/sdk);
* Хост приложений dotnet служит для запуска приложений .NET Core. Он выбирает среду выполнения, размещает ее, предоставляет политику загрузки сборок и запускает приложение. Этот же хост используется для запуска средств SDK очень похожим образом;
* .NET Core — это кроссплатформенная реализация .NET. Основные архитектурные особенности, уникальные для .NET Core, связаны с предоставлением реализаций для конкретных поддерживаемых платформ [9].

## Разработка модуля анализа

Поскольку в качестве основы для модуля анализа был выбран алгоритм Frequent Pattern-Growth Strategy, в модуле анализа должны иметься реализации его основных шагов: построения префиксного дерева популярных предметных наборов (FP-дерева) и извлечения частых наборов из FP-дерева.

Первоначально был произведен поиск готовых реализаций данных алгоритмов, однако это не дало результатов, поскольку таких реализаций очень мало, а те, которые удалось найти либо неработоспособны, либо не вписываются в концепцию проектируемой системы. Поэтому в рамках данной работы были разработаны собственные реализации алгоритмов, причем, в отличии от классического подхода, где данные действия выполняются последовательно друг за другом, в данной системе выполнение этих процедур производится независимо в разные моменты времени. Класс инкапсулирующий логику построения FP-дерева и извлечения из него частых наборов приведен в приложении А.

Отладка алгоритма производилась с использованием открытой базы данных транзакций «mushroom» взятой из FIMI Repository [10], а также библиотеки, pyfpgrowth написанной на языке python [11]. Для этого была установлена платформа Anaconda (см. рисунок 3.1), включающая в себя Python 2.7, более 150 предустановленных пакетов и Jupyter Notebook - интерактивная оболочка для языка программирования Python, позволяющая объединить код, текст и диаграммы.

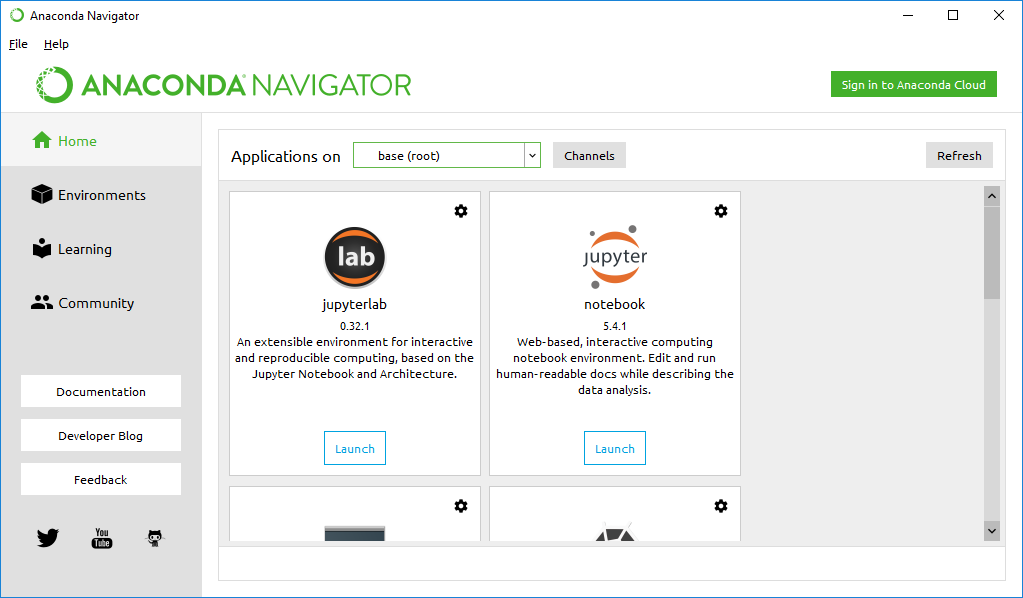


Рисунок 3.1 – Графический интерфейс Anaconda

Далее был создан python-скрипт, выполняющий поиск частых наборов с по 1000 транзакций в БД «mushroom» с минимальной поддержкой равной 1. Результат выполнения данного скрипта приведен на рисунке 3.2

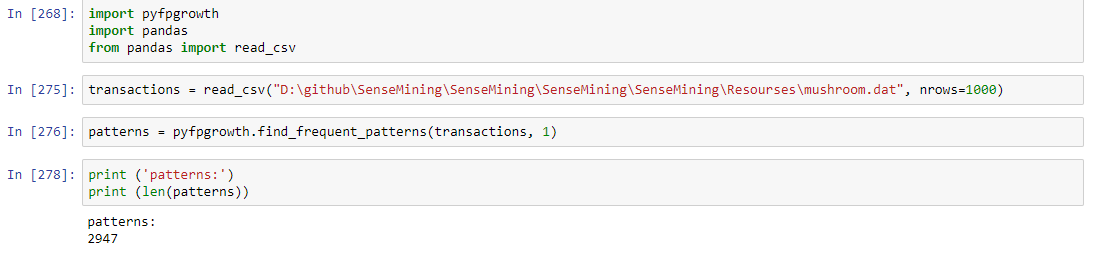


Рисунок 3.2 – Результат выполнения python-скрипта

На рисунке 3.3 приведен результат выполнения реализованного алгоритма поиска частых наборов для тех же транзакций, что и в предыдущем примере.

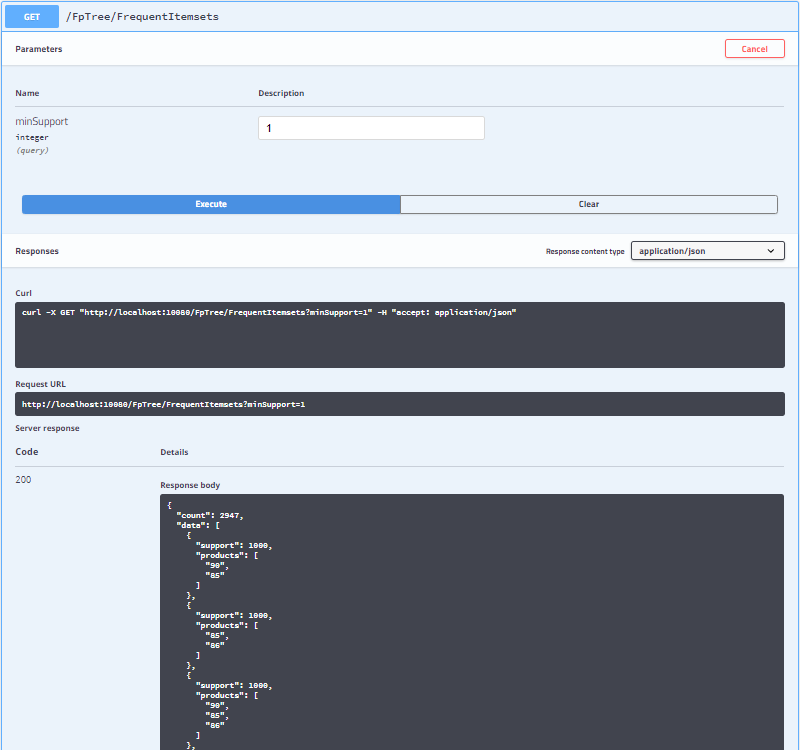


Рисунок 3.3 – Результат выполнения реализованного алгоритма

Из приведенных результатов видно, что в результате работы обоих алгоритмов для было получено одинаковое количество частых наборов, а именно 2947. Из этого можно сделать вывод о правильности работы реализованного алгоритма.

## Регистрация транзакций в системе

Поскольку в архитектуре системы определено, что источник данных (транзакций), связан с модулем предварительной обработки не на прямую, а посредством очереди сообщений, необходимо определить, с помощью каких инструментов будет реализована эта часть системы. Одним из самых популярных инструментов работы с очередями сообщений является RabbitMQ.

RabbitMQ — платформа, реализующая систему обмена сообщениями между компонентами программной системы (Message Oriented Middleware) на основе стандарта AMQP (Advanced Message Queuing Protocol). RabbitMQ выпускается под Mozilla Public License.

RabbitMQ создан на основе испытанной Open Telecom Platform, обеспечивающий высокую надёжность и производительность промышленного уровня и написан на языке Erlang.

RabbitMQ состоит из:

* Сервера RabbitMQ;
* Поддержки протоколов HTTP, XMPP и STOMP;
* Клиентских библиотек AMQP для Java и .NET Framework (поддержка других языков программирования реализована в ПО других производителей);
* Различных плагинов (таких как плагины для мониторинга и управления через HTTP или веб-интерфейс или плагин «Shovel» для передачи сообщений между брокерами).

Поддерживается горизонтальное масштабирование для построения кластерной архитектуры.

Имеется реализация клиентов для доступа к RabbitMQ для целого ряда языков программирования и платформ, широко используемых для веб-разработки: Java, .NET, Perl, Python, Ruby, PHP и других.

Сервер RabbitMQ по сути является менеджером очередей, который имеет следующие преимущества:

* В случае некорректного завершения работы сервера, данные в очереди не теряются. И при последующем запуске обработка продолжается с того места, где был обрыв;
* Распределить задачи на несколько очередей, т.е. создать распараллеливание на уровне сообщений;
* Если результат обработки не удовлетворяет, задачу можно послать в очередь повторно;
* Существует несколько режимов работы очереди: рассылка типа точка-точка(direct), рассылка сообщений по шаблону(topic), широковещательная рассылка сообщений(fanout);
* Возможность синхронизировать работу клиента и сервера, своего рода реализация RPC;
* Количество хранимых в очереди сообщений неограничено;
* Сервер сообщений может быть расположен удаленно как по отношению к продюсеру, так и по отношению к консьюмеру [12].

Таким образом для реализации очереди сообщений-транзакций будет использован брокер сообщений RabbitMQ.

## Поиск источника данных

Для наглядного примера работы системы необходим достаточно большой набор данных. Также предпочтительно, чтобы эти данные максимально соответствовали реальности.

Набор данных, подходящий для данной задачи был найден на kaggle (платформа для исследователей разных уровней, где они могут опробовать свои модели анализа данных на серьезных и актуальных задачах) [13]. Данный набор содержит в себе информацию о продажах сети продуктовых магазинов Market Grocery Company за 2018 год и включает более 6 миллионов транзакций. Данный набор представляет собой csv-файл объемом 518 мегабайт. Содержимое файла представлено на рисунке 3.4.

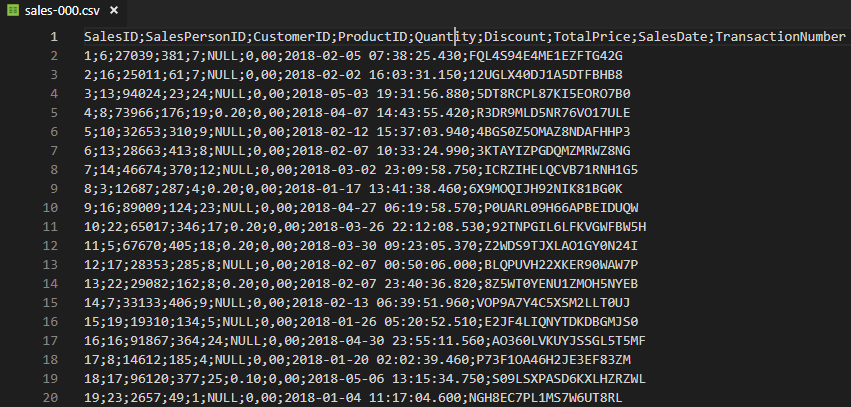


Рисунок 3.4 – Содержимое файла с продажами

Информация о покупателях, продавцах и самих товарах находится в отдельных csv-файлах. Содержимое файлов с информацией о товарах приведено на рисунках 3.5 и 3.6.

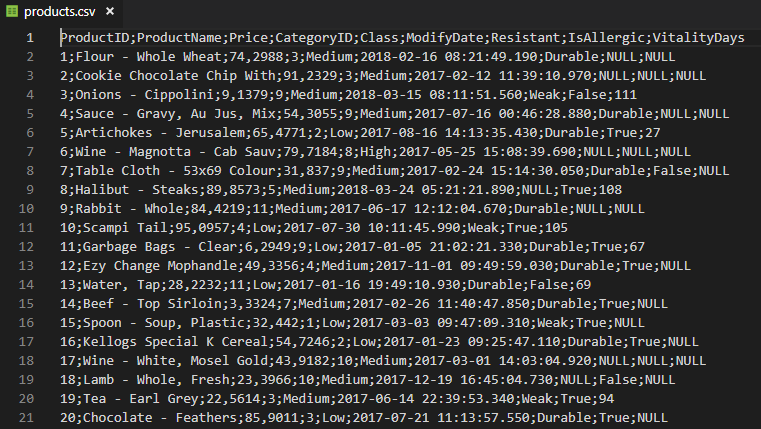


Рисунок 3.5 – Содержимое файла с продуктами

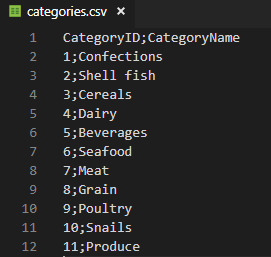


Рисунок 3.6 – Содержимое файла с категориями продуктов

Содержимое файлов с информацией о продавцах и покупателях приведено на рисунках 3.7 и 3.8.

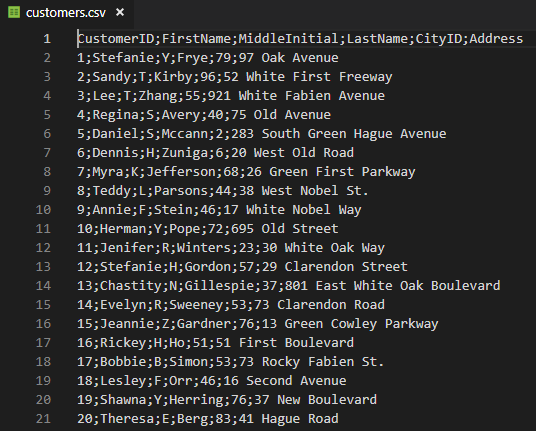


Рисунок 3.7 – Содержимое файла с покупателями

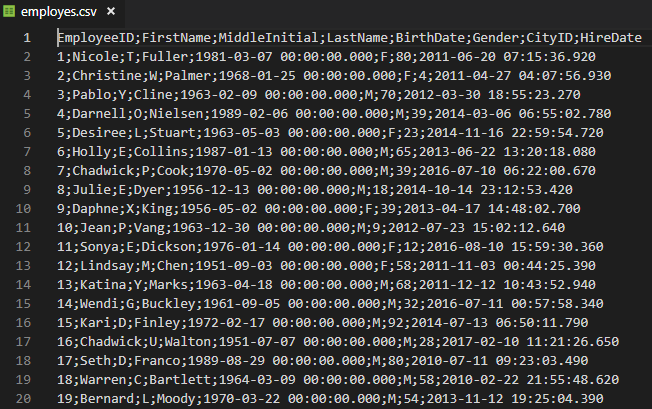


Рисунок 3.8 – Содержимое файла с продавцами

Для обработки csv файлов использовалась библиотека ChoETL.NETStandard. Данная библиотека позволяет считывать данные из csv-файлов, десериализуя их в пользовательские типы, а также формировать LINQ-запросы к этим данным не загружая в память файл целиком.

После десериализации транзакции передаются в очередь сообщений, к которой подключен обработчик транзакций. Графический интерфейс RabbitMQ в процессе обработки транзакций представлен на рисунке 3.9.

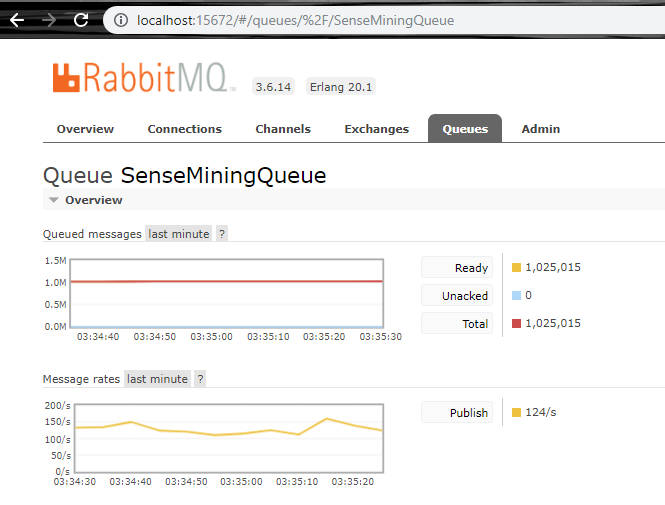


Рисунок 3.9 – Процесс обработки транзакций

Обработку и передачу транзакций в очередь сообщений производит отдельное консольное приложение. Код обработки и передачи транзакций приведен в приложении Б. Обработка транзакций заключается в группировке их по покупателю и времени покупки. Таким образом объединяются товары, вошедшие в одну покупку одного покупателя.

## Описание работы системы

Работа с системой производится двумя способами:

1. Посредством HTTP API;
2. Через очередь сообщений по протоколу AMQP (Advanced Message Queuing Protocol).

HTTP API, которое включает в себя следующий набор функций:

* Ручная регистрация транзакции в системе;
* Запуск обновления FP-дерева;
* Получение модели FP-дерева (Json);
* Получение частых наборов для заданного значения минимальной поддержки;
* Получение ассоциативных правил для заданного значения надежности.

Графический интерфейс Swagger вышеописанного API приведен на рисунке 3.10.

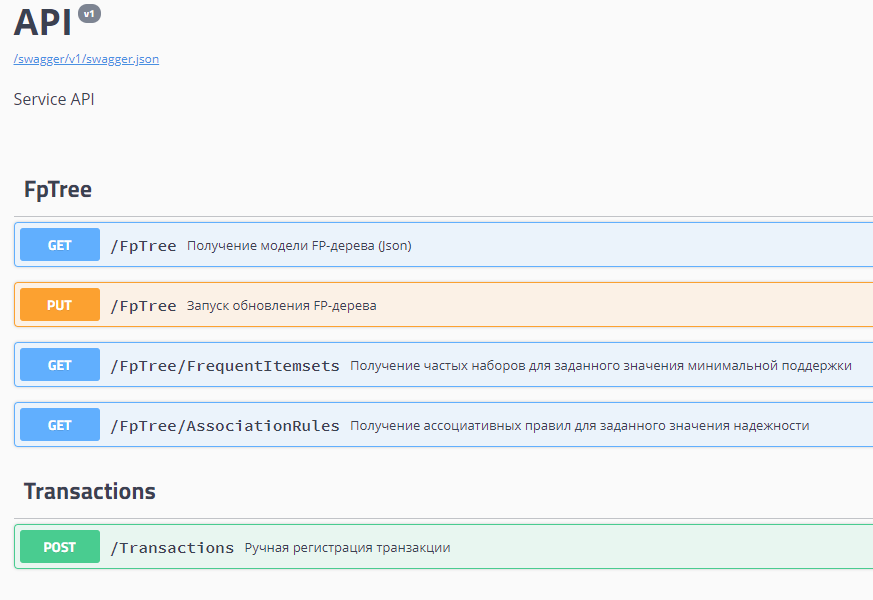


Рисунок 3.10 – Графический интерфейс Swagger

AMQP интерфейс системы представляет собой очередь SenseMiningQueue, в которую отправляет сообщения вышеописанное приложение – обработчик исходных данных транзакций. Это основной способ регистрации транзакций в системе и на его основе будет производиться интеграция данной системы с другими.

Запуск обработчика исходных транзакций вызвал помещение в очередь 1347150 транзакций (см. рисунок 3.11), обработка которых заняла порядка двух часов на четрыехъядерном процессоре (3,7 ГГц).

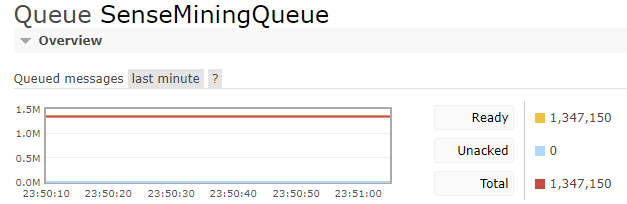


Рисунок 3.11 – Очередь сообщений

Далее был запущен алгоритм построения FP-дерева на основе этих транзакций. Выполнение данного алгоритма заняло 1732862 мс 29 мин (см. рисунок 3.12).

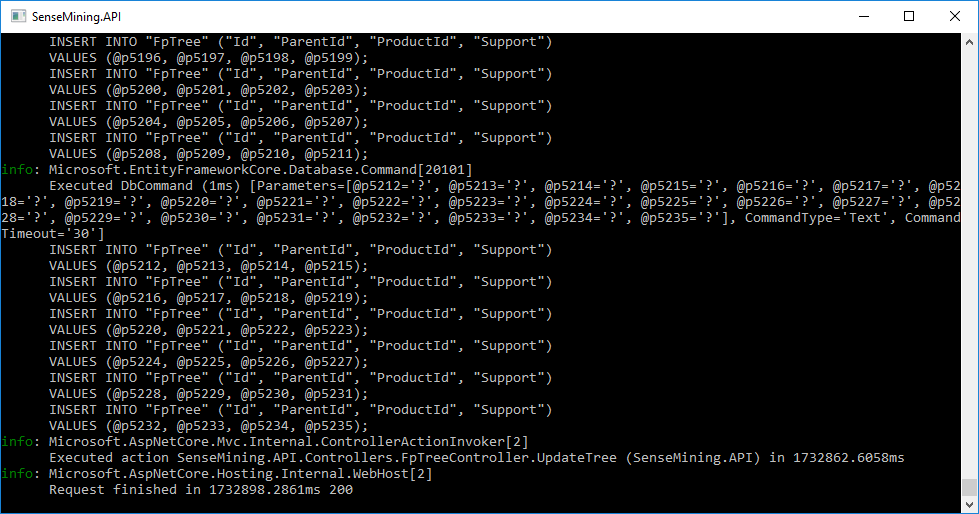


Рисунок 3.12 – Выполнение алгоритма построения FP-дерева

## Развертывание системы

Поскольку проектируемая система сама по себе не является конечным продуктом и предназначена для интеграции с другими системами, должна быть определена стратегия интеграции. Предполагается, что данная система будет являться частью сервис-ориентированной системы, развернутой под управлением Docker.

Docker – программное обеспечение с открытым исходным кодом для автоматизации развертывания приложений в среде виртуализации на уровне операционной системы. Docker позволяет отделить приложение от его инфраструктуры, упаковывая приложения со всеми зависимостями в стандартизованный модуль, называемый контейнером, который может быть перенесен на целевую операционную систему. Изоляция контейнеров позволяет запускать множество контейнеров одновременно на одной машине.

Docker предоставляет платформу и инструменты для управления жизненным циклом контейнеров:

* Разработка приложения и его вспомогательных компонентов с использованием контейнеров;
* Использование контейнера как единицы тестирования и поставки приложения;
* Развертывание приложения в релизной среде в виде контейнера или оркестирируемого сервиса.

Docker представляет собой клиент-серверное приложение, состоящее из следующих частей:

* Демон Docker – сервер, работающий в фоне, обрабатывающий запросы клиента и управляющий такими объектами как образы, контейнеры, сети и тома. Он также взаимодействует с другими демонами для управления работой сервисов.
* Клиент – приложение с консольным интерфейсом, позволяющее пользователям взаимодействовать с демоном Docker. Клиент может взаимодействовать более чем с одним демоном.
* Реестр – хранилище Docker образов. Может быть публичным (например, Docker Hub) или приватным.

Образ представляет собой шаблон с инструкциями для создания контейнера. Он доступен только для чтения.

Чаще всего образы бывают созданы на основе других образов. Например, можно создать образ, устанавливающий какое-либо приложение, основанные на образе с операционной системой. Таким образом, можно переиспользовать уже имеющиеся образы.

Каждый образ состоит из набора уровней. Docker использует UnionFile System для сочетания этих уровней в один образ. Union File System позволяет файлам и директориями из разных файловых систем (разным ветвям) прозрачно накладываться, создавая когерентную файловую систему.

Для создания собственного образа необходимо создать файл, называемый Dockerfile, определяющий шаги для создания образа и его запуска. Каждая такая инструкция создает новый слой в образе. В случае изменения и пересборки образа, пересобираются только измененные слои.

Вышеописанные особенности делают образы быстрыми и легковесными единицами развертывания приложения [14].

Таким образом разрабатываемая система должна будет распространяться в виде Docker-контейнера. Docker-файл API проекта приведен на рисунке 3.13.

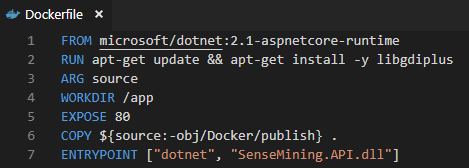


Рисунок 3.13 – Docker-файл API проекта

# Заключение

В результате выполнения данной работы была достигнута её основная цель проектирование и реализация основополагающих частей системы анализа фискальных данных, а также решены все поставленные задачи.

* Выбран оптимальный подход к решению задачи поиска ассоциативных правил;
* Описана базовая архитектура проектируемой системы;
* Спроектирована модель базы данных, с которой будет работать система;
* Разработан аналитический модуль системы;
* Выработана стратегия развертывания системы.

В результате выполнения данной работы был разработан прототип системы анализа фискальных данных, имеющий API для регистрации транзакций и получения аналитических данных в виде ассоциативных правил.

Необходимо отметить, что основная концепция данной системы, а также ее основополагающие элементы могут быть применены не только для анализа фискальных данных. Все это может быть применено в областях, где присутствует необходимость поиска ассоциативных правил, например, для задач оптимизации управления запасами.

# Список использованных источников

# Список литературы

|  |  |
| --- | --- |
| [1] | Федеральный закон от 22.05.2003 N 54-ФЗ (ред. от 03.07.2016) "О применении контрольно-кассовой техники при осуществлении наличных денежных расчетов и (или) расчетов с использованием электронных средств платежа". |
| [2] | Обязательные реквизиты кассового чека и БСО с 2017 года. [Электронный ресурс] URL: https://www.moysklad.ru/poleznoe/izmenenija-v-54-fz/novye-objazatelnye-rekvizity-chekov-kkt-i-bso (Дата обращения 12.12.2017). |
| [3] | Интеллектуальный анализ данных. [Электронный ресурс] URL: http://www.machinelearning.ru/wiki/index.php?title=Интеллектуальный\_анализ\_данных (Дата обращения 20.11.2017). |
| [4] | Яндекс.ОФД. [Электронный ресурс] URL: https://www.vedomosti.ru/business/articles/2017/08/30/731551-riteileri-uznayut-prodazhi (Дата обращения 15.12.2017). |
| [5] | Анализ рыночной корзины (Market Basket Analysis). [Элекронный ресурс] URL: https://basegroup.ru/community/glossary/market-basket (Дата обращения 15.12.2017). |
| [6] | Oded Maimon, Lior Rokach. Data Mining and Knowledge Discovery Handbook. Second Edition. Springer Science+Business Media, LLC 2005, 2010, 1285 с.. |
| [7] | Mohammed J. Zaki, Wagner Meira Jr. Data Mining and Analysis. Cambridge University Press, 2014, 593 с.. |
| [8] | Вершинин Е.В., Прокофьев М.Л., Афанасьев В.Р. Проектирование аналитической системы обработки фискальных данных // Вопросы радиоэлектроники 2019 .- № 3 .- С. 78 – 82.. |
| [9] | Руководство по .NET Core. [Электронный ресурс] URL: https://docs.microsoft.com/ru-ru/dotnet/core/ (Дата обращения 26.05.2018). |
| [10] | Frequent Itemset Mining Implementations Repository [Электронный ресурс] URL: http://fimi.ua.ac.be/ (Дата обращения 25.04.2018). |
| [11] | Orange3-Associate.[Электронный ресурс] URL: http://orange3-associate.readthedocs.io/en/latest (Дата обращения 25.04.2018). |
| [12] | RabbitMQ documentation. [Электронный ресурс] URL: https://www.rabbitmq.com/documentation.html (Дата обращения 28.05.2018). |
| [13] | Dataset from grocery market. [Электронный ресурс] https://www.kaggle.com/marcinex1423/salesdb-grocery. |
| [14] | Docker documentation. «Docker Overview». URI: https://docs.docker.com/engine/understanding-docker/ (дата обращения: 24.04.2018). |
| [15] | Д. О. Кириченко, М. А. Артемов ОПТИМИЗАЦИЯ ВХОДНЫХ ДАННЫХ В ЗАДАЧЕ ПОИСКА ШАБЛОНОВ И АССОЦИАТИВНЫХ ПРАВИЛ. ВЕСТНИК ВГУ, 2014, 63 с. |

# ПРИЛОЖЕНИЕ А

Листинг логики работы с FP-деревом

public class FpTreeService : IFpTreeService

{

private readonly IFpTreeProvider \_fpTreeProvider;

private readonly ITransactionsService \_transactionsService;

private readonly IProductsService \_productsService;

private readonly DatabaseContext \_dbContext;

private readonly CancellationToken \_cancellationToken;

public FpTreeService(IFpTreeProvider fpTreeProvider, ITransactionsService transactionsService,

IProductsService productsService, DatabaseContext dbContext, CancellationTokenSource cancellationTokenSource)

{

\_fpTreeProvider = fpTreeProvider;

\_transactionsService = transactionsService;

\_productsService = productsService;

\_dbContext = dbContext;

\_cancellationToken = cancellationTokenSource.Token;

}

/// <summary>

/// Обновление дерева последними зарегистрированными транзакциями

/// </summary>

public async Task UpdateTree()

{

var lastUpdate = await \_dbContext.UpdateHistory.OrderByDescending(a => a.CreationTime).FirstOrDefaultAsync(\_cancellationToken);

//Получаем элементы в порядке убывания поддержки

var order = await \_productsService.GetOrderedProducts();

if (lastUpdate == null) //FP-дерево не создано

{

await BuildNewTree(order);

}

else

{

var transactions = await \_transactionsService.GetLastTransactions(lastUpdate.CreationTime);

var tree = await \_fpTreeProvider.GetActualFpTree();

var root = tree.Single(a => a.Id == lastUpdate.RootId);

PushTransactions(transactions, order, root);

\_dbContext.UpdateHistory.Add(new FpTreeUpdateInfo(root.Id, DateTimeOffset.UtcNow));

}

await \_dbContext.SaveChangesAsync(\_cancellationToken);

}

/// <summary>

/// Постороение нового дерева

/// </summary>

/// <param name="order">Порядок сортировки</param>

private async Task BuildNewTree(List<Product> order)

{

var transactions = await \_transactionsService.GetLastTransactions(DateTimeOffset.MinValue);

var root = new Node();

PushTransactions(transactions, order, root);

\_dbContext.FpTree.Add(root);

\_dbContext.UpdateHistory.Add(new FpTreeUpdateInfo(root.Id, DateTimeOffset.UtcNow));

}

/// <summary>

/// Извлечение частых наборов из актуального дерева

/// </summary>

/// <param name="minSupport">Минимальная поддержка</param>

public async Task<List<FrequentItemsetModel>> ExtractFrequentItemsets(int minSupport)

{

var products = await \_productsService.GetOrderedProducts();

var tree = await \_fpTreeProvider.GetActualFpTree();

var result = new List<FrequentItemsetModel>();

//var resultSet = new HashSet<FrequentItemsetModel>();

foreach (var product in products)

{

//Условный базис

var conditionalBase = tree.Where(a => a.ProductId == product.Id)

.Select(a => a.PathToRoot.Select(n =>

new

{

ProductId = n.Node.ProductId.Value,

n.Node.Product,

n.Node.Support,

ConditionalSupport = a.Support,

Item = n

}));

//Суммарная поддержка каждого элемента в условном базисе

var totalSups = conditionalBase.SelectMany(a => a)

.GroupBy(a => a.ProductId).ToDictionary(a => a.Key, a => a.Sum(p => p.ConditionalSupport));

foreach (var branch in conditionalBase)

{

foreach (var item in branch)

{

var support = totalSups[item.ProductId];

//Фильтрация редких элементов

if (support < minSupport)

continue;

var set = CollectFrequentItemsets(item.Item, support, totalSups);

//Фильтрация повторов

if (result.Any(a => a.Products.Intersect(set).Count() == set.Count()))

continue;

result.Add(new FrequentItemsetModel

{

Support = support,

Products = set

});

}

}

}

return result;

}

/// <summary>

/// Вставка в дерево списка транзакций

/// </summary>

/// <param name="transactions">Транзакции</param>

/// <param name="order">Порядок</param>

/// <param name="root">Корневой узел дерева</param>

private void PushTransactions(List<Transaction> transactions, List<Product> order, Node root)

{

var comparer = new ProductsComparer(order); //Компаратор элементов

foreach (var transaction in transactions)

{

var orderedItems = transaction.Items.OrderBy(a => a.ProductId, comparer);

var tempRoot = root;

Node tempNode;

foreach (var transactionItem in orderedItems)

{

var aNode = new Node(transactionItem.ProductId, tempRoot.Id, 1);

if ((tempNode = tempRoot.Children.FirstOrDefault(c => c.ProductId == aNode.ProductId)) != null)

{

tempNode.Support++;

tempRoot = tempNode;

}

else

{

tempRoot.Children.Add(aNode);

tempRoot = aNode;

}

}

}

}

/// <summary>

/// Выборка частых наборов из ветви условного базиса

/// </summary>

private IEnumerable<Product> CollectFrequentItemsets(ConditionalTreeItem root, int support, Dictionary<Guid, int> totalSups)

{

var node = root;

do

{

if (totalSups.TryGetValue(node.Node.ProductId.Value, out int s) && s < support) continue;

yield return node.Node.Product;

} while ((node = node.Next) != null);

}

}

# ПРИЛОЖЕНИЕ Б

Листинг логики обработки и отправки в очередь транзакций

public class TransactionsDbProcessor

{

private readonly ImporterOptions \_options;

private readonly IPublishEndpoint \_publishEndpoint;

private readonly Dictionary<int, string> \_procucts;

private readonly Dictionary<int, string> \_procuctCategories;

public TransactionsDbProcessor(ImporterOptions options, IPublishEndpoint publishEndpoint)

{

\_options = options;

\_publishEndpoint = publishEndpoint;

\_procucts = new Dictionary<int, string>();

\_procuctCategories = new Dictionary<int, string>();

}

public async Task Start()

{

using (var reader = new ChoCSVReader<CategoryModel>(Path.Combine(\_options.ResourcesPath, "categories.csv"))

.WithFirstLineHeader())

{

foreach (var categoryModel in reader)

{

\_procuctCategories.Add(categoryModel.CategoryID, categoryModel.CategoryName);

}

}

using (var reader = new ChoCSVReader<ProductModel>(Path.Combine(\_options.ResourcesPath, "products.csv"))

.WithFirstLineHeader())

{

foreach (var productModel in reader)

{

\_procucts.Add(productModel.ProductID,

$"{productModel.ProductName};{\_procuctCategories[productModel.CategoryID]}");

}

}

using (var reader =

new ChoCSVReader<SalesRowModel>(Path.Combine(

"salesDB\_grocery\_market", "sales.csv"))

.WithFirstLineHeader())

{

var groupsByCustomer = reader

.Where(a => a.SalesDate.HasValue)

.GroupBy(a => a.CustomerID);

var groupsByCustomerByDate = groupsByCustomer

.Select(a => new

{

CustomerID = a.Key,

Days = a

.GroupBy(b => b.SalesDate.Value.Date)

.Where(d => d.Count() > 1)

}).Where(a => a.Days.Any()).ToArray();

foreach (var groupByCustomer in groupsByCustomerByDate)

{

foreach (var groupByDate in groupByCustomer.Days)

{

await \_publishEndpoint.Publish<ITransactionMessage>(new

{

Items = groupByDate.Select(a => \_procucts[a.ProductID])

});

}

}

}

}

}