ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ»
(Н И У « Б е л Г У »)

ИНСТИТУТ ИНЖЕНЕРНЫХ ТЕХНОЛОГИЙ И ЕСТЕСТВЕННЫХ НАУК

КАФЕДРА МАТЕМАТИЧЕСКОГО ОБЕСПЕЧЕНИЯ И АДМИНИСТРИРОВАНИЯ ИНФОРМАЦИОННЫХ СИСТЕМ

РАЗРАБОТКА АЛГОРИТМА ПОИСКА И АНАЛИЗА СВОЙСТВ ОБЪЕКТОВ НА МЕСТНОСТИ С ПРИМЕНЕНИЕМ КВАДРОДЕРЕВА

Выпускная квалификационная работа
обучающегося по направлению подготовки
02.03.03 Математическое обеспечение и администрирование информационных систем
очной формы обучения,
группы 07001302
Помельникова Антона Владимировича

Научный руководитель:
к. т. н, доцент Михелев В.М.

БЕЛГОРОД 2017

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 4
ГЛАВА 1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ 6
1.1 Основные понятия предметной области 6
1.2 Методы иерархического разбиения пространства 8
1.2.1 kD-деревья 9
1.2.2 BSP-деревья 11
1.2.3 Interval Tree 12
1.2.4 R-деревья 13
1.2.5 Дерево квадрантов 15
1.3 Обзор алгоритмов кластеризации данных 17
1.3.1 Меры расстояний 18
1.3.2 Алгоритмы иерархической кластеризации 20
1.3.3 Объединение кластеров 21
1.4 Выбор инструментальных средств разработки 23
ГЛАВА 2 ПРОЕКТИРОВАНИЕ ВЕБ-ПРИЛОЖЕНИЯ 28
2.2 Проектирование структуры веб-приложения 28
2.3 Проектирование базы данных 29
2.4 Определение требований к алгоритму 31
ГЛАВА 3 РАЗРАБОТКА ВЕБ-ПРИЛОЖЕНИЯ 33
3.1 Разработка алгоритма поиска и анализа свойств объектов на местности 33
3.2 Разработка информационного обеспечения 36

3.3 Разработка веб-приложения 37
ГЛАВА 4 ИНТЕГРАЦИЯ И ТЕСТИРОВАНИЕ ПО 44
4.1 Интеграция ПО в геоинформационную систему «Активист» 44
4.2 Тестирование алгоритма поиска и анализа свойств объектов на местности 46
ЗАКЛЮЧЕНИЕ 50
СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 50
ПРИЛОЖЕНИЕ 1 53
ПРИЛОЖЕНИЕ 2 54

ВВЕДЕНИЕ

Трудно представить себе современный мир без интерактивных географических карт на компьютерах и мобильных устройствах в наши дни. В настоящее время навигация и нахождение нужных мест на карте стали намного проще благодаря развитию различных геоинформационных систем.

Однако удобство использование интерактивных карт на компьютерах и мобильных устройствах целиком и полностью зависит от правильного отображения нужных данных на экране пользователя. Перерисовка десятков тысяч объектов при навигации в пользовательском интерфейсе системе занимает длительное время. Распространенным решением проблемы отображения географических объектов является их предварительная кластеризация с целью отображения мест скопления объектов, а не отдельных объектов.

Поэтому все еще остается актуальной задачей анализа и обобщения картографических данных с целью донесения до конечного пользователя той информации, в которой он нуждается.

Актуальность темы выпускной квалификационной работы заключается в необходимости оптимизации производительности отображения географических объектов в картографических приложениях для совершенствования пользовательского опыта взаимодействия.

Цель данной выпускной квалификационной работы заключается в оптимизации производительности отображения большого количества объектов на интерактивной карте в веб-приложении используя разработанный алгоритм.

Для выполнения поставленной цели необходимо решить следующие задачи:

1. Проанализировать предметную область.

2. Спроектировать веб-приложение, определить функциональные требования к алгоритму.

3. Разработать веб-приложение и алгоритм.

4. Интегрировать и протестировать полученный алгоритм.

В главе «Анализ предметной области» будут описаны основные понятия предметной области, проведен обзорный анализ методов иерархического разбиения двухмерного пространства, а также обзор алгоритмов кластеризации данных. Кроме того, будет проведен выбор инструментальных средств разработки.

В главе «Проектирование веб-приложения» будет описано проектирование основных составных частей веб-приложения и алгоритма.

В главе «Разработка веб-приложения» будет описана последовательность основных этапов разработки веб-приложения и алгоритма.

В главе «Интеграция и тестирование ПО» будут приведены данные тестирования алгоритма, а также описаны результаты интеграции разработанного алгоритма в систему.

В заключении сделан вывод о степени достижения поставленных целей и задач.

Данная выпускная квалификационная работа содержит 52 страницы, 20 рисунков, 6 формул, 10 листингов и 2 приложения.

ГЛАВА 1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Основные понятия предметной области

В настоящее время интерактивные географические карты используются повсеместно для навигации по городу, прокладывания маршрутов следования, поиска адресов и геолокации. Практически все геоинформационные сервисы, предоставляющие данные работают по технологии клиент-сервер. Клиент в данном случае – это чаще всего веб-браузер или мобильное приложение. В качестве сервера выступает программа, которая собирает, обрабатывает и хранит данные, а также отвечает на запросы клиента.

Поскольку понятия, используемые в названии темы могут иметь разное значение, то стоит определить их в контексте данной работы.

Под *местностью* понимается двухмерное пространство географической интерактивной карты. Поскольку карта является интерактивной, т.е. подразумевает взаимодействие с пользователем, то на ней должны присутствовать элементы управления, такие как зум – увеличение или уменьшение масштаба карты, курсор, линейка – возможность отмерить расстояние на карте и другие.

Рассмотрим, что же такое объект на местности. Под *объектом* понимается любая географическая точка, которая имеет свои координаты (широта, долгота) и представляет собой маркер, расположенный на карте. Далее по тексту понятия объект на карте и маркер будут иметь равносильное значение.

Квадродерево – иерархическая структура данных, в котором у каждого внутреннего узла ровно четыре потомка. Далее по тексту квадродерево будет более подробно описано.

Представим себе следующую ситуацию. Пользователь некоего сервиса по поиску недвижимости хочет посмотреть все предложения по продаже квартир в городе N. Он заходит на сервис с помощью веб-браузера, выбирает все доступные предложения и нажимает кнопку «найти». В этот момент перед ним открывается карта города N со всеми квартирами, представленными на карте в виде маркеров. По мере появления на карте все большего количества маркеров карта просто перестает быть видимой и браузер начинает тормозить из-за большой нагрузки на него. Поскольку современные клиентские устройства хоть и обладают хорошей производительностью, с такой нагрузкой из-за особенностей браузерных технологий они не могут справиться.

Итак, возникла проблема отображения большого количества объектов на местности. Отсюда можно сделать два вывода – объекты надо как-то обрабатывать перед тем, как показать их пользователю и решить каким именно образом их обрабатывать.

Наиболее частым решением данной проблемы является предварительная кластеризация объектов на карте в некие группы – кластеры, которые объединяют в себе некоторое количество объектов. Существует много алгоритмов кластеризации данных, использующих разные метрики, но в каждом случае выбор алгоритма зависит от потребностей пользователя. Объединение объектов может происходить по разным признакам.

В данной работе предполагается, что объекты будут иметь некоторые свойства, по которым можно разбить все объекты на группы. Под свойством будем понимать некую категорию или разновидность маркера. В реальных системах это может быть категория типа «ресторан», «кафе» или «гостиница».

Помимо выбора алгоритма кластеризации необходимо выбрать структуру данных, которая позволяла бы максимально эффективно быстро находить ее элементы и позволяла бы делать выборку элементов не потребляя при этом большого количества ресурсов.

В данной работе предполагается разработать веб-приложение для апробации алгоритма поиска и анализа объектов на местности. Почему именно веб-приложение? Ответ лежит на поверхности – так как большинство клиентских устройств взаимодействует с сервисами посредством веб-браузера.

Перед тем как погрузиться в проектирование будущего алгоритма и веб-приложения для его реализации, стоит рассмотреть методы иерархического разбиения пространства и существующие алгоритмы кластеризации данных, чтобы выбрать из них подходящий для решения нашей задачи.

1.2 Методы иерархического разбиения пространства

При работе двумерным пространством часто возникает необходимость в различных запросах, связанных с пространственным расположением объектов в пространстве.

Типичными примерами подобных запросов являются:

- определение объектов, попавших в заданную область пространства
- определение объектов, объектов, пересекаемых заданным лучом
- определение ближайшего объекта, пересекаемого заданным лучом
- определение столкновения объектов между собой
- определение N ближайших объектов к заданному

Подобные запросы обычно имеют сложность $O(n)$, где n – общее количество объектов в сцене. Однако, в ряде случаев сложность может достигать $O(n^2)$ или даже еще более высокой степени.

Из этого видно, что для больших сцен метод «грубой силы» (т.е. прямого перебора) просто неприемлем из-за своих больших затрат.

Таким образом, возникает необходимость в методах с сублинейной сложностью (от общего количества объектов), а в идеале - когда сложность

метода прямо пропорциональна количеству объектов, найденных данным запросом (такие методы называются *output sensitive*).

Стандартным приемом, позволяющим заметно снизить сложность запросов о взаимном расположении объектов в пространстве, являются различные типы так называемых пространственных индексов (*spatial index*) [10].

Пространственный индекс – это некоторая структура данных (чаще всего иерархическая), строящаяся обычно на этапе подготовки сцены.

Далее мы рассмотрим основные типы пространственных индексов.

1.2.1 kD-деревья

Так называемые kD-деревья на каждом шаге производят разбиение только вдоль одной плоскости. Обычно в качестве плоскости разбиения выбирается плоскость, перпендикулярная оси, вдоль которой ячейка имеет наибольший размер (см. рис. 1.1).

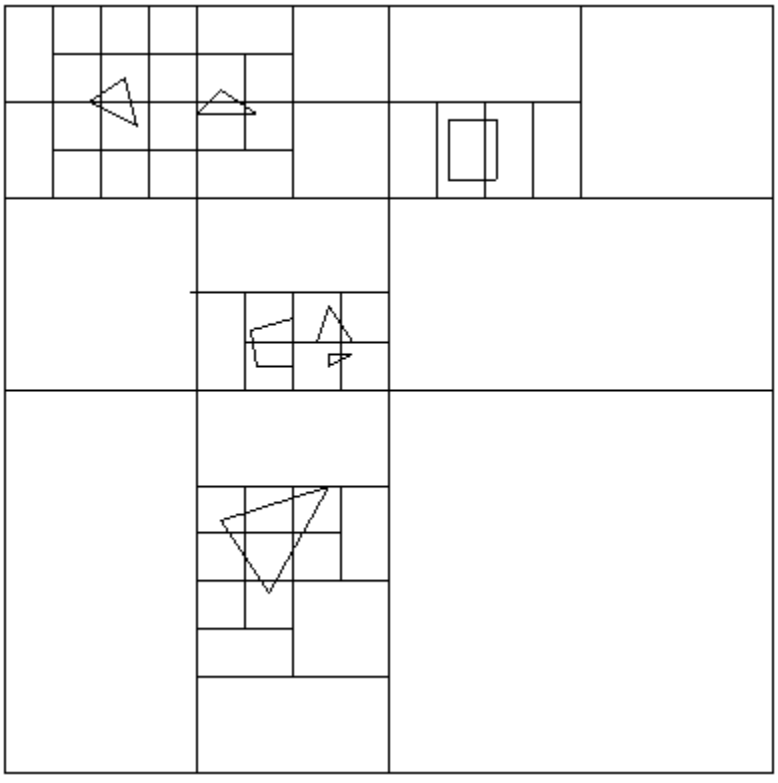


Рис. 1.1. Разбиение ячейки для kD-дерева.

При использовании такой стратегии разбиения сохраняется все его преимущества – адаптивный характер разбиения и временные затраты *O(log n)*.

За счет этого подхода, у нас гарантированно будут получаться ячейки, размеры которых вдоль различных осей, не будут сильно отличаться.

Кроме того, у kD-деревьев есть еще одна интересная особенность, не обязательно каждый раз разбивать ячейку на две равные половинки. Вместо этого можно также использовать адаптивный алгоритм – выбирать такое положение разбивающей плоскости вдоль оси разбиения, при котором в получившихся половинках будет примерно одинаковое число объектов.

При реализации kD-деревьев возможны два разных подхода к случаю, когда разбивающая плоскость пересекает один из объектов сцены.

Первый подход заключается в разбиении исходного объекта на две части, каждая из которых идет в соответствующий узел.

Второй подход вместо разбиения объекта просто помещает ссылку на него в оба соответствующих списка.

Основным плюсом первого подхода является однозначная классификация объектов по узлам. При этом списки для дочерних узлов заданного узла никогда не пересекаются и возможно очень легкое построение частичного упорядочивания объектов – всегда можно утверждать, что ни один из объектов из одного (более удаленного) узла никогда не сможет закрыть никакого объекта из другого узла.

Кроме того, при втором подходе необходимо отслеживать какие объекты уже были обработаны (так как мы можем снова встретить ссылку на этот объект в списке другого узла).

И то, и другое довольно неудобно, кроме того, разбиение объектов заметно увеличивает общее число объектов, что также довольно плохо с точки зрения производительности [7].

1.2.2 BSP-деревья

Еще одним из распространенных пространственных индексов являются BSP (*Binary Space Partitioning*) деревья (деревья двоичного разбиения пространства).

При построении этого дерева выбирается некоторая плоскость и все объекты разбиваются на два кластера, в зависимости от положения по отношению к этой плоскости. Объекты, пересекаемые плоскостью, разбиваются вдоль нее.

К каждому из двух полученных множеств вновь применяется подобный процесс разбиения и так далее (см. рис. 1.2)

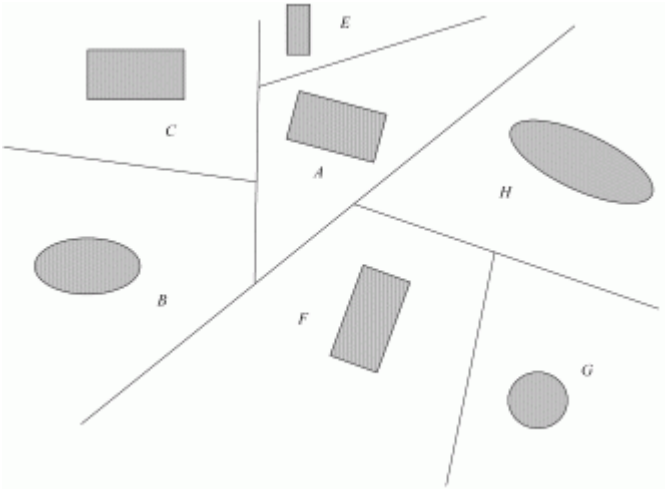


Рис. 1.2. Построение BSP-дерева.

Таким образом BSP-дерево фактически является разновидностью kD-дерева (или наоборот), однако здесь нет жесткого правила выбора разбивающей плоскости.

Таким образом BSP-дерево фактически является разновидностью kD-дерева (или наоборот), однако здесь нет жесткого правила выбора разбивающей плоскости. (см. рис. 1.3)

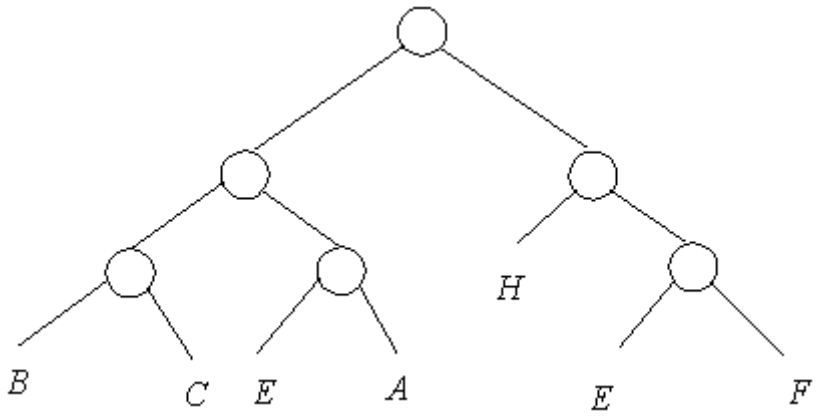


Рис. 1.3. BSP-дерево

При построении BSP-деревьев возможно два критерии прекращения рекурсии.

1. В списке узла осталось не более одной грани.
2. Все оставшиеся в списке узла грани являются частью границ выпуклого многогранника.

Оба этих критерия связаны с тем, что BSP-дерево для граней задумывалось как механизм сортировки граней. В этом случае ясно, что критерием остановки должна быть однозначная упорядочиваемость всех граней узла по отношению к произвольному положению камеры.

При этом видно, что для критерия 1 оставшиеся в узле грани (либо одна, либо ни одной) однозначно упорядочиваются.

Для второго критерия хотя такого однозначно упорядочивания может и не быть, но из лицевых граней узла, ни одна не может закрывать другую, тем самым они все равно упорядочиваемым [1].

1.2.3 Interval Tree

В самом простейшем случае одномерных интервалов (сегментов) *interval tree* выступает в качестве пространственного индекса, используемого для

работы с большим количеством (возможно пересекающихся между собой) отрезков. Существует два варианта построения одномерных *interval tree*.

В первом случае мы выбирает некоторое значение x_c и делим весь набор отрезков на три части - *L* (все отрезки целиком лежащие левее x_c), *R* (все отрезки целиком лежащие правее x_c) и *C* (все отрезки, пересекающие x_c). На рис. 1.4 изображено разбиение отрезков для построения дерева.

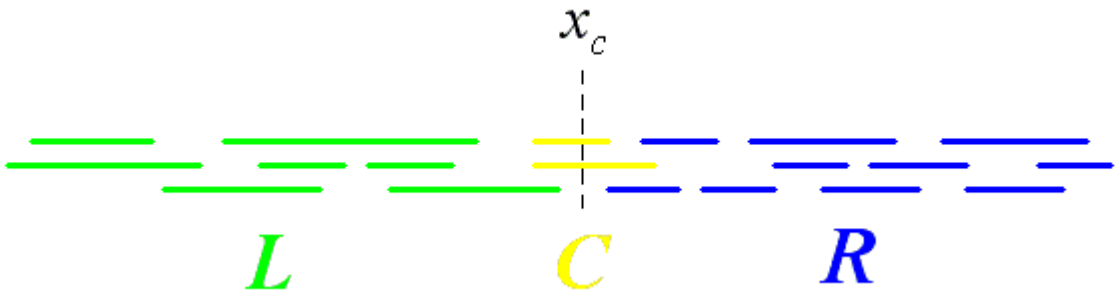


Рис. 1.4. Разбиение отрезков для построение *interval tree*.

После этого список *C* связывается с текущим узлом дерева, а по *L* и *R* строятся левое и правое поддеревья аналогичным образом.

Второй вариант использует обычное бинарное дерево для упорядочивания отрезков по их началу, но при этом для каждого узла дерева хранится максимальная правая координата для всех отрезков соответствующего поддерева. Оба этих способа обеспечивают быстрое построение дерева по набору отрезков и логарифмическую скорость локализации интересующих отрезков (кроме вырожденных случаев).

Оба описанных подхода легко обобщаются и на многомерный случай – в этом случае вместо отрезков у нас выступают многомерные AABB.

1.2.4 R-деревья

Еще одним вариантом деревьев, используемых для поиска на плоскости или в пространстве, являются так называемые R-деревья.

R-дерево строится по ограничивающим прямоугольным параллелепипедам (AABB) исходных объектов и во многом напоминает B-деревья.

Каждый узел R-дерева порядка *(m,M)* содержит от *m* до *M* записей (корень дерева может содержать до двух записей), с каждой записью связан AABB для всех объектов соответствующего поддерева и ссылка на узел этого поддерева.

В отличии от B-деревьев AABB, соответствующие записям (поддеревьям) могут пересекаться между собой. В остальном работа с R-деревьями сильно напоминает работу с B-деревьями – при добавлении новой записи может произойти переполнение узла и тогда необходимо его разбить на два (вынеся ссылку на добавленную запись наверх), при уничтожении записи может возникнуть необходимость объединения узла с соседним (в этом случае наверху удаляется ссылка).

Для случая переполнения узла существует две стратегии разбиения всех AABB по двум получившимся узлам. Обе они основаны на выборе двух AABB, задающих изначальные группы. После чего каждый из оставшихся AABB добавляется к той группе, для которой его добавление дает минимальное увеличение итоговой площади получающегося AABB.

Первая стратеги (квадратичная) заключается в выборе той пары AABB b_i и b_j, которые дают наибольшую площадь, если они окажутся в одной группе.

Вторая стратегия (линейная) заключается в выборе b_i и b_j, обладающих наибольшим расстоянием вдоль каждой оси. На рисунке 1.5 показан пример R-дерева.

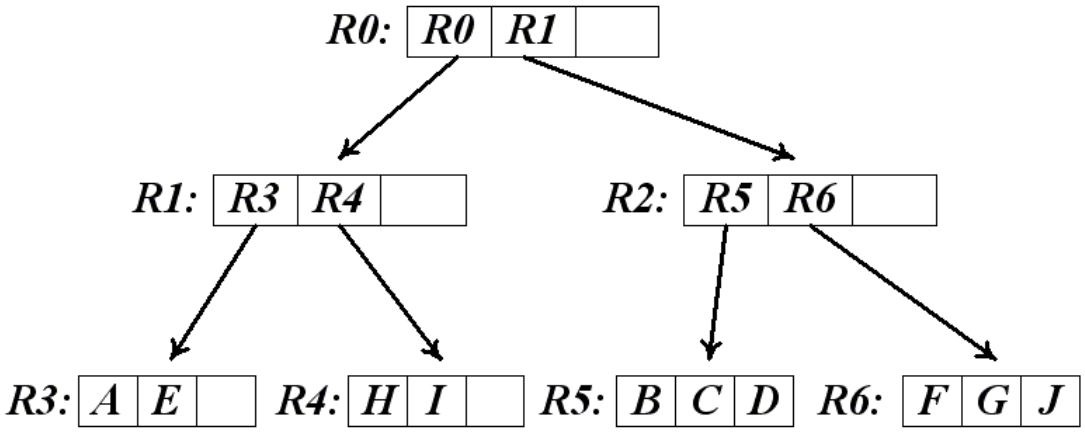


Рис. 1.5. R-дерево.

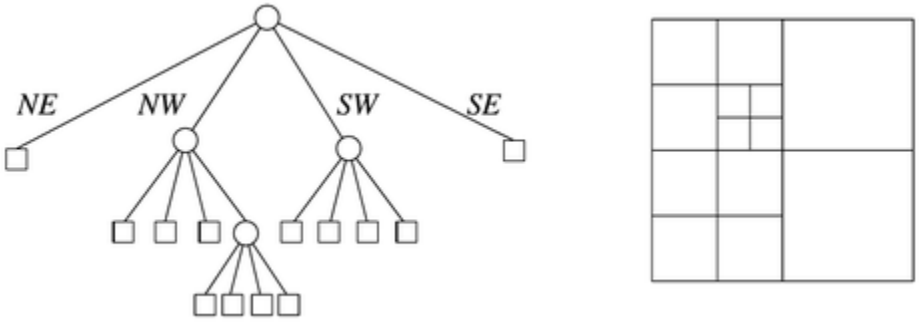
Как можно заметить R-деревья не гарантируют ларифмическую скорость (в худшем случае), но на реальных данных ведут себя хорошо. Типичным применением для R-деревьев являются различные СУБД по пространственным данным (например различные геоинформационные системы) [1].

1.2.5 Дерево квадрантов

Деревья квадрантов часто используются для рекурсивного разбиения двухмерного пространства по 4 квадранта (области). Области представляют собой квадраты. Аналогичное разбиение пространства известно как Q-дерево. Общие черты разных видов деревьев квадрантов:

- разбиение пространства на адаптирующиеся ячейки
- максимально возможный объём каждой ячейки,
- соответствие направления дерева пространственному разбиению.

Фрагмент дерева квадрантов представлен на рис. 1.6.



.

Рис. 1.6. Фрагмент дерева квадрантов

Квадродерево позволяет производить динамическое перестроение кластеров без перезапуска алгоритма на всем множестве объектов. Деревья квадрантов могут быть классифицированы в соответствии с типом данных, который они представляют – пространством, точками, прямыми, кривыми. Также их можно разделить по зависимости формы дерева от порядка обработки данных. Рассмотрим некоторые виды деревьев квадрантов.

Region quadtree. Деревья квадрантов, разбивающие пространство, представляют какую-либо часть двухмерного пространства разбивая его на 4 квадранта, субквадранты и так далее, причем каждый лист дерева соответствует определенной области. У каждого узла дерева либо 4 потомка, либо их нет вовсе (у листьев). Деревья квадрантов, разбивающие пространство, не являются деревьями в полной мере, поскольку положение субквадрантов не зависит от данных. Более правильное название – префиксные деревья.

Если дерево используется для представления множества точек (например, широты и долготы положений каких-либо городов), области разбиваются до тех пор, пока листья будут содержать не более одной точки.

Point quadtree. Также существуют деревья квадрантов, которые хранят информацию о точках. Это такая разновидность бинарных деревьев, используемых для хранения информации о точках на плоскости. Время обработки при использовании таких деревьев составляет $O(\log n)$, что очень эффективно в сравнении упорядоченных точек на плоскости.

Структура узла. Узел дерева квадрантов, хранящего информацию о точках плоскости, аналогичен узлу бинарного дерева лишь с той оговоркой, что узел первого имеет четыре потомка (по одному на каждый квадрант) вместо двух («правого» и «левого»). Ключ узла состоит из двух компонент (для координат *x* и *y*). Таким образом, узел дерева содержит следующую информацию:

- 4 указателя: *quad[‘NW’]*, *quad[‘NE’]*, *quad[‘SW’]*, и *quad[‘SE’]*,
- точка, описываемая следующим образом: ключ *key*, обычно выражает координаты *x* и *y*, значение *value*, например, имя.

Все рассмотренные выше иерархические структуры имеют между собой целый ряд общих черт.

- Они строятся как рекурсивное разбиение исходного множества объектов.
- Средние временные затраты для них составляют *O(log n)*,
- С каждым узлом связан список содержащихся в нем объектов и ограничивающее тело,
- Большинство из них устойчивы по отношению к локальным изменениям геометрии сцены,
- Они достаточно легко обеспечивают частичное упорядочивание группы объектов [4].

1.3 Обзор алгоритмов кластеризации данных

Кластеризация (или кластерный анализ) — это задача разбиения множества объектов на группы, называемые кластерами. Внутри каждой группы должны оказаться «похожие» объекты, а объекты разных группы должны быть как можно более отличны. Главное отличие кластеризации от классификации состоит в том, что перечень групп четко не задан и определяется в процессе работы алгоритма [9].

Кластерный анализ выполняет следующие основные задачи:

- Разработка типологии или классификации.
- Исследование полезных концептуальных схем группирования объектов.
- Порождение гипотез на основе исследования данных.
- Проверка гипотез или исследования для определения, действительно ли типы (группы), выделенные тем или иным способом, присутствуют в имеющихся данных.

Применение кластерного анализа в общем виде сводится к следующим этапам:

1. Отбор выборки объектов для кластеризации.
2. Определение множества переменных, по которым будут оцениваться объекты в выборке. При необходимости – нормализация значений переменных.
3. Вычисление значений меры сходства между объектами.
4. Применение метода кластерного анализа для создания групп сходных объектов (кластеров).
5. Представление результатов анализа.

После получения и анализа результатов возможна корректировка выбранной метрики и метода кластеризации до получения оптимального результата.

1.3.1 Меры расстояний

Так, перед нами встала задача определения «схожести» объектов. Сначала нужно составить вектор характеристик для каждого объекта. Вектор представляет собой набор числовых значений, например, рост-вес человека. Также существуют алгоритмы, которые работают с качественными характеристиками.

Определив вектор характеристик, можно провести нормализацию, чтобы все компоненты давали одинаковый вклад при расчете «расстояния». В процессе нормализации все значения приводятся к некоторому диапазону, например, [-1, -1] или [0, 1].

Для каждой пары объектов измеряется «расстояние» между ними – степень похожести. Существует множество метрик, вот лишь основные из них.

Евклидово расстояние. Наиболее распространенная функция расстояния. Представляет собой геометрическим расстоянием в многомерном пространстве:

$$\rho(x, x') = \sqrt{\sum_i^n (x_i - x'_i)^2} \quad (1.1)$$

где: x – значение i -свойства объекта x, а x_i' – значение i – свойства объекта x;

Квадрат евклидова расстояния. Применяется для придания большего веса более отдаленным друг от друга объектам. Это расстояние вычисляется следующим образом:

$$\rho(x, x') = \sum_i^n (x_i - x'_i)^2 \quad (1.2)$$

где: x – значение i -свойства объекта x, а x_i' – значение i – свойства объекта x;

Расстояние городских кварталов (манхэттенское расстояние). Это расстояние является средним разностей по координатам. В большинстве случаев эта мера расстояния приводит к таким же результатам, как и для обычного расстояния Евклида. Однако для этой меры влияние отдельных больших разностей (выбросов) уменьшается (т.к. они не возводятся в квадрат) [10]. Формула для расчета манхэттенского расстояния:

$$\rho(x, x') = |x_i - x'_i| \quad (1.3)$$

Расстояние Чебышева. Это расстояние может оказаться полезным, когда нужно определить два объекта как «различные», если они различаются по какой-либо одной координате. Расстояние Чебышева вычисляется по формуле:

$$\rho(x, x') = \max(|x_i - x'_i|) \quad (1.4)$$

Степенное расстояние. Применяется в случае, когда необходимо увеличить или уменьшить вес, относящийся к размерности, для которой соответствующие объекты сильно отличаются. Степенное расстояние вычисляется по следующей формуле:

$$\rho(x, x') = \sqrt[r]{\sum_i^n (x_i - x'_i)^p} \quad (1.5)$$

где: r и p – параметры, определяемые пользователем. Параметр p ответственен за постепенное взвешивание разностей по отдельным координатам, параметр r ответственен за прогрессивное взвешивание больших расстояний между объектами. Если оба параметра – r и p – равны двум, то это расстояние совпадает с расстоянием Евклида.

Выбор метрики полностью лежит на исследователе, поскольку результаты кластеризации могут существенно отличаться при использовании разных мер [8].

1.3.2 Алгоритмы иерархической кластеризации

Среди алгоритмов иерархической кластеризации выделяются два основных типа: восходящие и нисходящие алгоритмы. Нисходящие алгоритмы работают по принципу «сверху-вниз»: в начале все объекты помещаются в один кластер, который затем разбивается на все более мелкие

кластеры. Более распространены восходящие алгоритмы, которые в начале работы помещают каждый объект в отдельный кластер, а затем объединяют кластеры во все более крупные, пока все объекты выборки не будут содержаться в одном кластере. Таким образом строится система вложенных разбиений. Результаты таких алгоритмов обычно представляют в виде дерева – дендрограммы. Классический пример такого дерева – классификация животных и растений [9].

Для вычисления расстояний между кластерами все чаще пользуются двумя расстояниями: одиночной связью или полной связью.

Алгоритмы квадратичной ошибки. Задачу кластеризации можно рассматривать как построение оптимального разбиения объектов на группы. При этом оптимальность может быть определена как требование минимизации среднеквадратической ошибки разбиения:

$$e^2(X, L) = \sum_{j=1}^{K} \sum_{i=1}^{n_j} \left\| x_i^{(j)} - c_j \right\|^2 \quad (1.6)$$

где: c_j — «центр масс» кластера j (точка со средними значениями характеристик для данного кластера).

Алгоритмы квадратичной ошибки относятся к типу плоских алгоритмов. Самым распространенным алгоритмом этой категории является метод k-средних. Этот алгоритм строит заданное число кластеров, расположенных как можно дальше друг от друга. В качестве критерия остановки работы алгоритма обычно выбирают минимальное изменение среднеквадратической ошибки. К недостаткам данного алгоритма можно отнести необходимость задавать количество кластеров для разбиения.

1.3.3 Объединение кластеров

В случае использования иерархических алгоритмов встает вопрос, как объединять между собой кластера, как вычислять «расстояния» между ними. Существует несколько метрик:

Одиночная связь (расстояния ближайшего соседа). В этом методе расстояние между двумя кластерами определяется расстоянием между двумя наиболее близкими объектами (ближайшими соседями) в различных кластерах. Результирующие кластеры имеют тенденцию объединяться в цепочки.

Полная связь (расстояние наиболее удаленных соседей). В этом методе расстояния между кластерами определяются наибольшим расстоянием между любыми двумя объектами в различных кластерах (т.е. наиболее удаленными соседями). Этот метод обычно работает очень хорошо, когда объекты происходят из отдельных групп. Если же кластеры имеют удлиненную форму или их естественный тип является «цепочечным», то этот метод непригоден.

Невзвешенное попарное среднее. В этом методе расстояние между двумя различными кластерами вычисляется как среднее расстояние между всеми парами объектов в них. Метод эффективен, когда объекты формируют различные группы, однако он работает одинаково хорошо и в случаях протяженных («цепочечного» типа) кластеров.

Взвешенное попарное среднее. Метод идентичен методу невзвешенного попарного среднего, за исключением того, что при вычислениях размер соответствующих кластеров (т.е. число объектов, содержащихся в них) используется в качестве весового коэффициента. Поэтому данный метод должен быть использован, когда предполагаются неравные размеры кластеров.

Невзвешенный центроидный метод. В этом методе расстояние между двумя кластерами определяется как расстояние между их центрами тяжести.

Взвешенный центроидный метод (медиана). Этот метод идентичен предыдущему, за исключением того, что при вычислениях используются веса

для учета разницы между размерами кластеров. Поэтому, если имеются или подозреваются значительные отличия в размерах кластеров, этот метод оказывается предпочтительнее предыдущего [7].

Рассмотрев основные понятия предметной области, методы иерархического разбиения пространства и сделав обзор алгоритмов кластеризации, перейдем к проектированию веб-приложения и алгоритма, где и выберем подходящий для нас метод разбиения пространства и алгоритм кластеризации.

1.4 Выбор инструментальных средств разработки

Изучив и проанализировав теоретическую часть, необходимо выбрать инструментальные средства для разработки алгоритма и веб-приложения.

Так как в браузере основной язык разработки является JavaScript, то выберем его для написания нашей клиентской части приложения. Стоит поближе рассмотреть, что же такое JavaScript, и какие у него возможности.

Итак, JavaScript — это мультипарадигменный язык программирования, который поддерживает такие стили как: объектно-ориентированный, императивный и функциональны. Является реализацией языка ECMAScript (стандарт ECMA-262).

JavaScript используется в основном в клиентской части веб-приложений: клиент-серверных программ, в которых в качестве клиента выступает браузер, а в качестве сервера – веб-сервер, имеющие распределенную логику между клиентом и сервером. Преимуществом такого метода является то, что клиенты независимы от операционной системы пользователя, таким образом веб-приложения являются кроссплатформенными [15].

За свою простоту и широкие возможности применения выбор языка программирования для проекта остановился на JavaScript.

Последние тренды клиентской и серверной веб-разработки предлагают стек технологий основанный на языке JavaScript. Такой стек называется MEAN — (аббревиатура от MongoDB, Express.js, Angular.js, Node.js) — комплекс серверного программного обеспечения, который используется для веб-разработки кроссплатформенных приложений. Поскольку все компоненты набора поддерживают программирование на JavaScript, и серверная и клиентская часть MEAN-приложений может быть написана на этом языке программирования.

Компоненты:

- MongoDB — документоориентированная СУБД;
- Express.js — каркас веб-приложений, работающий поверх Node.js;
- Angular.js — MVC-фреймворк для фронтенда, интерфейсной части веб-приложения, работающей в браузере;
- Node.js — JavaScript платформа для серверной разработки.

Разберем каждую технологию по отдельности.

MongoDB — документоориентированная система управления базами данных (СУБД) с открытым исходным кодом, не требующая описания схемы таблиц. Классифицируется как NoSQL, использует JSON-подобные документы и схему базы данных. Написана на языке C++.

MongoDB реализует новый подход к построению баз данных, где нет таблиц, схем, запросов SQL, внешних ключей и многих других вещей, которые присущи объектно-реляционным базам данных.

В отличие от реляционных баз данных MongoDB предлагает документо-ориентированную модель данных, благодаря чему MongoDB работает быстрее, обладает лучшей масштабируемостью, ее легче использовать.

Но, даже учитывая все недостатки традиционных баз данных и достоинства MongoDB, важно понимать, что задачи бывают разные и методы их решения бывают разные. В какой-то ситуации MongoDB действительно

улучшит производительность вашего приложения, например, если надо хранить сложные по структуре данные. В другой же ситуации лучше будет использовать традиционные реляционные базы данных. Кроме того, можно использовать смешанный подход: хранить один тип данных в MongoDB, а другой тип данных – в традиционных БД.

Формат данных в MongoDB. Для хранения в MongoDB применяется формат, который называется BSON (БиСон) или сокращение от binary JSON.

BSON позволяет работать с данными быстрее: быстрее выполняется поиск и обработка. Хотя надо отметить, что BSON в отличие от хранения данных в формате JSON имеет небольшой недостаток: в целом данные в JSON-формате занимают меньше места, чем в формате BSON, с другой стороны, данный недостаток с лихвой окупается скоростью. [12]

Кроссплатформенность. MongoDB написана на C++, поэтому ее легко портировать на самые разные платформы. MongoDB может быть развернута на платформах Windows, Linux, MacOS, Solaris. Можно также загрузить исходный код и самому скомпилировать MongoDB, но рекомендуется использовать библиотеки с официального сайта.

Документы вместо строк. Если реляционные базы данных хранят строки, то MongoDB хранит документы. В отличие от строк документы могут хранить сложную по структуре информацию. Документ можно представить как хранилище ключей и значений. Ключ представляет простую метку, с которым ассоциирована определенная часть данных.

Коллекции. Если в традиционном мире SQL есть таблицы, то в мире MongoDB есть коллекции. И если в реляционных БД таблицы хранят однотипные жестко структурированные объекты, то в коллекции могут содержать самые разные объекты, имеющие различную структуру и различный набор свойств.

Простота в использовании. Отсутствие жесткой схемы базы данных и в связи с этим потребности при малейшем изменении концепции хранения

данных пересоздавать эту схему значительно облегчают работу с базами данных MongoDB и дальнейшим их масштабированием. Кроме того, экономится время разработчиков. Им больше не надо думать о пересоздании базы данных и тратить время на построение сложных запросов.

MongoDB имеет драйверы для интеграции с Node.js и со многими другими серверными платформами.

Express. Это каркас web-приложений для Node.js, реализованный как свободное и открытое программное обеспечение под лицензией MIT. Он спроектирован для создания веб-приложений и API. Де-факто является стандартным каркасом для Node.js. Автор фреймворка, TJ Holowaychuk, описывает его как созданный на основе написанного на языке Ruby каркаса Sinatra, подразумевая, что он минималистичен и включает большое число подключаемых плагинов. Express является backend'ом для программного стека MEAN, вместе с базой данных MongoDB и каркасом AngularJS для frontend'a.

AngularJS. JavaScript-фреймворк с открытым исходным кодом. Предназначен для разработки одностраничных приложений. Его цель — расширение браузерных приложений на основе MVC-шаблона, а также упрощение тестирования и разработки.

Фреймворк адаптирует и расширяет традиционный HTML, чтобы обеспечить двустороннюю привязку данных для динамического контента, что позволяет автоматически синхронизировать модель и представление. В результате AngularJS уменьшает роль DOM-манипуляций и улучшает тестируемость. Отделение DOM-манипуляции от логики приложения, что улучшает тестируемость кода.

Angular придерживается MVC-шаблона проектирования и поощряет слабую связь между представлением, данными и логикой компонентов. Используя внедрение зависимости, Angular переносит на клиентскую сторону такие классические серверные службы, как видозависимые контроллеры.

Следовательно, уменьшается нагрузка на сервер и веб-приложение становится легче [14].

NodeJS. Программная платформа, основанная на движке V8 (транслирующем JavaScript в машинный код), превращающая JavaScript из узкоспециализированного языка в язык общего назначения. Node.js добавляет возможность JavaScript взаимодействовать с устройствами ввода-вывода через свой API (написанный на C++), подключать другие внешние библиотеки, написанные на разных языках, обеспечивая вызовы к ним из JavaScript-кода. Node.js применяется преимущественно на сервере, выполняя роль веб-сервера. Также существует возможность разрабатывать на Node.js и десктопные оконные приложения (при помощи NW.js, AppJS или Electron для Linux, Windows и Mac OS) и даже программировать микроконтроллеры.

В основе Node.js лежит событийно-ориентированное и асинхронное (или реактивное) программирование с неблокирующим вводом/выводом.

Кроме того, все механизмы обработки запросов и прочих операций ввода/вывода (I/O) построены на событиях, и это означает, что в Node.js нет никакого способа, чтобы заблокировать работающий в данный момент поток. Каждая операция в Node.js выполняется асинхронно, что является огромным преимуществом, особенно если код должен быть построен на операциях ввода-вывода: чтение дисков, подключение к базе данных, веб-сервисы и т.д.

Производительность в такой системе гораздо выше, чем, если использовалась многопоточная модель (multi-threaded blocking model). Примером многопоточной модели является веб-сервер Apache и Nginx [13].

Вышеупомянутый стек технологий предлагает отличный каркас для быстрого развертывания приложения на любой платформе.

ГЛАВА 2 ПРОЕКТИРОВАНИЕ ВЕБ-ПРИЛОЖЕНИЯ

2.2 Проектирование структуры веб-приложения

Современное веб-приложение предполагает в своей основе использование клиент-серверной архитектуры. В данном случае под клиентом подразумевается веб-приложение в браузере и является точкой входа в программу. При обращении к определенному адресу (URL) приложение загружается в браузер пользователя и начинает свою работу. При каком-либо пользовательском действии на сервер отправляется запрос, который обрабатывается и возвращается ответ. Также если запрашиваются какие-то данные, то происходит обращение к базе данных (БД). На рис. 2.1 показана схема такого подхода взаимодействия.

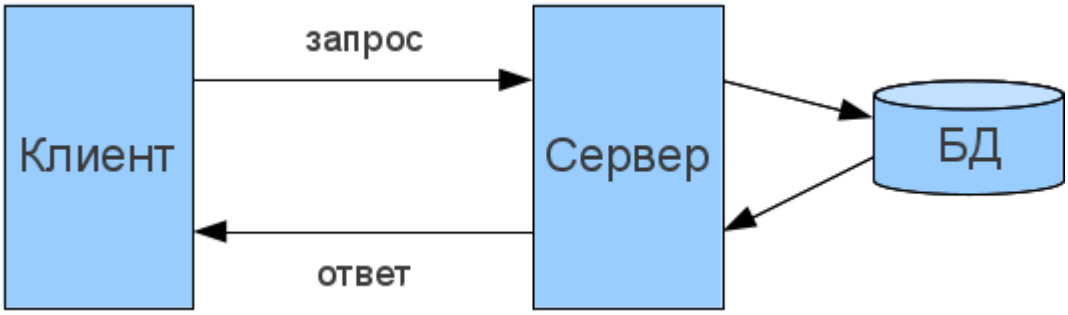


Рис. 2.1. Архитектура «Клиент-сервер»

Ранее рассматривался стек MEAN, который используется в современных веб-приложениях. Его мы и будем использовать в нашем приложении. Прежде всего начнем с описания шаблона проектирования MVC, который используется в нашем случае в клиентской части.

Model-View-Controller (MVC) — схема разделения данных приложения, пользовательского интерфейса и управляющей логики на три отдельных компонента: модель, представление и контроллер – таким образом, что модификация каждого компонента может осуществляться независимо [17].

- Модель (Model) предоставляет данные и реагирует на команды контроллера, изменяя свое состояние.
- Представление (View) отвечает за отображение данных модели пользователю, реагируя на изменения модели.
- Контроллер (Controller) интерпретирует действия пользователя, оповещая модель о необходимости изменений.

На рис. 2.2 можно ознакомиться со схемой паттерна MVC.

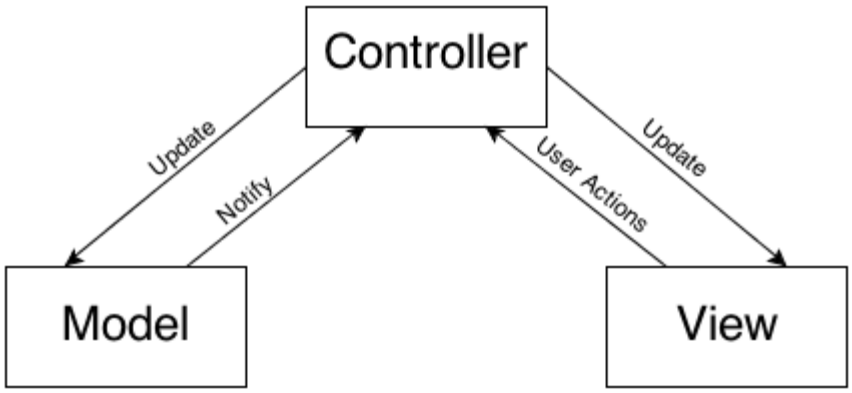


Рис. 2.2. Схема шаблона проектирования MVC

Итак, определившись со архитектурой клиентской части приложения, начнем проектировать серверную часть. Она будет отвечать за организацию маршрутизации (routing), а также обращаться к базе данных. Поскольку проектируемое веб-приложение подразумевает собой прототип реального веб-приложения, то не идет привязка к конкретным сущностям. Рассмотрим далее проектирование базы данных.

2.3 Проектирование базы данных

Инфологическое проектирование БД. Перед тем как проектировать базу данных, стоит учесть особенность нереляционной природы MongoDB. В ней отсутствует ограничения внешнего ключа, но существует возможность автоматического связывания между документами. Инфологическая модель в первую очередь связывается с возможность отображения связей между

отдельными сущностями, представляемыми в базе данных. Инфологическая модель БД представлена на рис. 2.3.

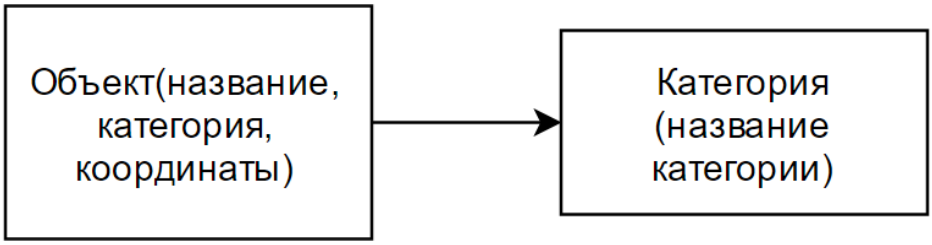


Рис. 2.3. Инфологическая модель БД

Так как база данных является тестовой, то есть не предполагает реальных данных, то ее схема довольно простая. В качестве внешнего ключа в таблице «Объект» выступает поле «Код категории». Поле «Координаты» будет представлять собой составное значение – широта и долгота.

Даталогическое проектирование. Даталогическая (концептуальная или логическая) модель БД — модель логического уровня, представляющая собой отображение логических связей между элементами данных, независимо от их содержания и среды хранения.

На данном этапе даталогического проектирования строится логическая структура базы данных нашего веб-приложения. При этом происходит преобразование исходной инфологической модели (рис.2.3) в модель данных, которая поддерживается конкретной СУБД. После этого производится проверка адекватности даталогической модели, отображаемой предметной области. Ниже на рис. 2.4 представлена датологическая модель базы данных.

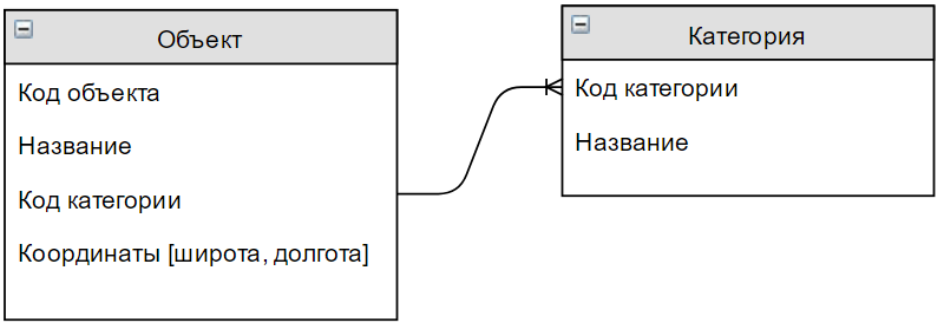


Рис. 2.4. Датологическая модель БД

Физическое проектирование БД. Разработка физической модели БД — процесс подготовки описания реализации базы данных на вторичных запоминающих устройствах. Физическая модель БД отображает таблицы и их названия, поля таблиц, их типы и размеры, и связи между таблицами.

Важным преимуществом MongoDB является поддержка геопространственных индексов и формата GeoJSON для представления геоданных. Для хранения координат будем использовать геопространственные индексы 2dsphere, которые имеют формат записи [долгота, широта]. Что это нам дает? Возможность осуществлять поиск в некотором радиусе, поиск пересечений, а также любые другие запросы, связанные с выборкой в некоторых пространственных координатах. На рис. 2.5 представлена физическая модель базы данных.

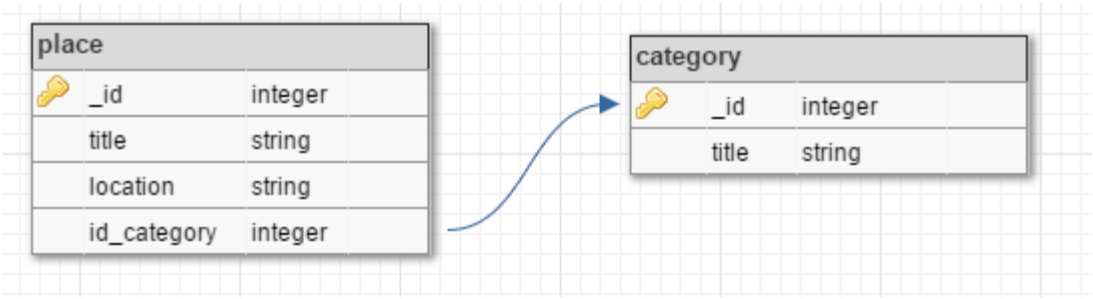


Рис. 2.5. Физическая модель БД

Далее необходимо разработать требования и функционал алгоритма поиска и анализа объектов на местности с учетом спроектированного приложения.

2.4 Определение требований к алгоритму

Разрабатываемый алгоритм основан на дереве квадрантов, поэтому все операции кластеризации, поиска и выборки будут происходить с использованием этой структуры данных. Рассмотрим подробнее как будет разбиваться пространство на квадранты. Так, будем использовать точечное

квадродерево, каждая точка которого будет представлять собой объект на карте. Так, в одном квадранте не может содержаться более четырех точек.

Следующая точка, попавшая в этот квадрант становится тем граничным условием, при котором текущий квадрант делится на 4 части. Так, деление пространства происходит рекурсивно пока последняя точка не будет помещена на плоскость.

Определим функциональные требования к алгоритму. Алгоритм должен:

- принимать в качестве исходных данных массив объектов с координатами.
- строить квадродерево из полученных объектов
- кластеризовать объекты на всех возможных уровнях приближения карты.
- динамически перестраивать квадродерево при удалении или добавлении объектов на местность.

Разобравшись с определением функциональных требований, перейдем к разработке веб-приложения и алгоритма.

ГЛАВА 3 РАЗРАБОТКА ВЕБ-ПРИЛОЖЕНИЯ

3.1 Разработка алгоритма поиска и анализа свойств объектов на местности

Перейдем к разработке алгоритма поиска и анализа свойств объектов. В основе алгоритма лежит построение квадродерева. Начнем рассмотрение с функции инициализации дерева. На листинге 3.8 приведен код функции.

Листинг 3.8. Функция инициализации квадродерева

```
function init(options) {
  if (! ('top' in options) || ! ('left' in options) || ! ('bottom' in options) || ! ('right' in options)) {
      throw new    Error('not enough init options');
  }
  opts = {
      top: options.top,
      left: options.left,
      bottom: options.bottom,
      right: options.right,
      max_per_cell: options.max_per_cell || 2,
      max_depth: options.max_depth || 4,
      debug_mode: options.debug_mode || false
  };

  tree = new Node(opts.top, opts.left, opts.bottom, opts.right, 0);
}
init(options);
```

Здесь проверяются начальные опции дерева и происходит создание корневого узла дерева. Рассмотрим далее основные методы вставки, деления и выборки над узлами квадродерева.

Операция вставки проверяет, в какой квадрант вставлять очередную точку. Если точка за пределами границ квадранта, то ничего не делаем. Если же дерево содержит дочерние узлы, то делегируем вставку дочерним узла. Если вставка невозможна в узлы, то мы передаем управление методу деления пространства на 4 области. На листинге 3.9 приведен код функции вставки.

Листинг 3.9. Функция вставки элемента

```
this.insert = function insert(itemForIndex){
   if (! this.containsPoint(itemForIndex.x, itemForIndex.y)) {
     return false;}
  if (this.nodes) {
     var nodes = this.nodes;
     if (nodes[TOP_LEFT].insert(itemForIndex)) return true;
     if (nodes[TOP_RIGHT].insert(itemForIndex)) return true;
     if (nodes[BOTTOM_LEFT].insert(itemForIndex)) return true;
     if (nodes[BOTTOM_RIGHT].insert(itemForIndex)) return true;
     if (opts.debug_mode) {
        throw new Error('item was not inserted into any node: ' +
itemAsString(itemForIndex));}
     return false;}
  if (this.items.length < opts.max_per_cell || this.depth >= opts.max_depth) {
     this.items.push(itemForIndex);
     return true;
  }
  this.divide();
  return this.insert(itemForIndex);
};
```

Разберем теперь функцию деления узла квадродерева на 4 квадранта. Если в текущем узле недостаточно места чтобы вставить еще одну точку, то текущий узел становится родительским узлом и все его точки переносятся в дочерние узлы, иначе говоря происходит деление. На листинге 3.10 приведен код функции деления.

Листинг 3.10. Код функции деления

```
this.divide = function divide() {
  var nodes = this.nodes = {},
      halfWidth = (this.right - this.left) / 2,
      halfHeight = (this.bottom - this.top) / 2,
      debug = opts.debug_mode,
      resInsert;
  nodes[TOP_LEFT] = new Node(this.top, this.left, this.top + halfHeight, this.left + halfWidth, depth + 1);
  nodes[TOP_RIGHT] = new Node(this.top, this.left + halfWidth, this.top + halfHeight, this.right, depth + 1);
  nodes[BOTTOM_LEFT] = new Node(this.top + halfHeight, this.left, this.bottom, this.left + halfWidth, depth + 1);
  nodes[BOTTOM_RIGHT] = new Node(this.top + halfHeight, this.left + halfWidth, this.bottom, this.right, depth + 1);

  for (var i = 0, items = this.items, l = items.length; i < l; i ++) {
      resInsert = this.insert(items[i]);
      if (debug && ! resInsert) {
          throw new Error('Item not inserted while dividing: ' + itemAsString(items[i]));
      }}
  this.items = [];
  return true;
};
```

Далее разберем как будут объединяться точки в кластеры. Мы будем использовать сеточный метод кластеризации, при котором карта делится на квадраты определенного размера (меняющегося для каждого уровня масштабирования), и в каждом таком квадрате маркеры группируются. Для определенного маркера создается кластер, к которому добавляются маркеры в границах квадрата кластера. Процесс повторяется до тех пор, пока все маркеры не будут включены в ближайшие сеточные кластеры маркеров с учетом уровня масштабирования карты.

Рассмотрев основные моменты реализации алгоритма, настало время сделать тестовые замеры скорости рендеринга и обработки точек на клиенте, а также интегрировать алгоритм в геоинформационную систему. Это мы сделаем в следующей главе.

3.2 Разработка информационного обеспечения

Итак, приступим к реализации тестовой базы данных. Для начала подключим модуль Mongoose для работы с MongoDB. Mongoose представляет специальную ODM-библиотеку (Object Data Modelling) для работы с MongoDB, которая позволяет сопоставлять объекты классов и документы коллекций из базы данных. Подключение библиотеки в Node.js показано на листинге 3.1.

Листинг 3.1. Подключение библиотеки Mongoose

```
var mongoose    = require('mongoose');
```

Далее создадим схему нашей коллекции (таблицы). Описание структуры коллекции происходит с помощью ключевого слова new Schema. На листинге 3.2 показано создание схемы коллекции Place.

Листинг 3.2. Создание схемы коллекции Place

```
var PlaceSchema = new Schema({
  title: {type: String, required: true},
  category: {type: String,required: true},
  location: {type: [Number], required: true} // [Long, Lat]
});
PlaceSchema.index({location: '2dsphere'});
```

Заметим, что мы указали создание индекса для документа (поля) location как 2dsphere. Ранее было оговорено почему мы это сделали. Создание второй коллекции «Category» опустим. Нас интересует то, как мы свяжем эти две коллекции. Используя функциональность DBRef, мы можем установить автоматическое связывание между документам. На листинге 3.3 показан пример связывания с помощью DBRef.

Листинг 3.3. Автоматическое связывание с помощью DBRef

```
moscow = ({«title»: «Moscow», «location»: [55.6767, 37.5198], category: new
DBRef(category, category._id)})
```

Установим соединение с базой данных с помощью метода mongoose.connect(). На листинге 3.4 показано подключение к БД.

Листинг 3.4. Подключение к базе данных

```
mongoose.connect(«mongodb://localhost/QuadTreeCluster»);
```

Примеры поиска и вставки в базу данных будут приведены в разделе «Разработка веб-приложения».

3.3 Разработка веб-приложения

Так как разрабатываемое приложение имеет архитектуру клиент-сервер, начнем прежде всего с серверной части. Рассмотрим архитектуру серверной части. Точкой входа в приложение является файл server.js – в нем содержится

подключение необходимых модулей, подключение к базе данных, конфигурация фреймворка Express, а также запуск самого сервера. На листинге 3.5 приведен фрагмент файла с созданием и инициализацией сервера.

Листинг 3.5. Создание и инициализация сервера

```
var express        = require('express');
var mongoose       = require('mongoose');
var port           = process.env.PORT || 3000;
var morgan         = require('morgan');
var bodyParser     = require('body-parser');
var app            = express();
var Place          = require('./app/model.js');
// Sets the connection to MongoDB
mongoose.connect(«mongodb://localhost/QuadTreeCluster»);
require('./app/routes.js')(app);
app.listen(port);
console.log('App listening on port ' + port);
```

Чтобы запустить приложение, достаточно консоли набрать команду nodemon server.js. В консоли должно отобразиться сообщение об успешном запуске. На рис. 3.1 показан ход запуска приложения в командной строке.

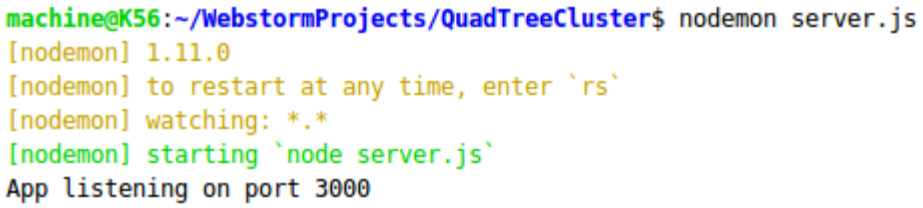


Рис. 3.1. Запуск приложения

Далее необходимо настроить маршрутизацию приложения (routing). Реализуем маршрутизацию, используя возможности Express. Файл routes.js отвечает за роутинг внутри приложения. Используя методы протокола HTTP

– GET и POST зададим обработчики функций соответственно. На листинге 3.5 приведен фрагмент двух обработчиков.

Листинг 3.5. Обработчики маршрутизации

```
module.exports = function(app) {
app.get('/places', function(req, res){
    // Uses Mongoose schema to run the search (empty conditions)
    var query = Place.find({});
    query.exec(function(err, places){
        if(err)
            res.send(err);
            res.json(places);
    });
 });
 // POST Routes
 app.post('/places', function(req, res){
         var newPlace= new Place(req.body);
    newPlace.save(function(err){
        if(err)
            res.send(err);
            res.json(req.body);
    });
 });
};
```

Рассмотрим подробнее что здесь происходит. При запросе в браузере методом GET, обработчик принимает функцию обратного вызова, в которой вызывается метод find(). В переменную query записывается команда поиска всех документов из базы данных. Далее выполняется метод exec(), который исполняет запрос в базу данных. Также он принимает функцию обратного

вызова в качестве аргумента и по условию успешного завершения возвращает в результат выборку, иначе посылает сообщение об ошибке. В случае запроса POST выполняются аналогичные действия, за исключением что происходит запись в базу данных, а не выборка.

Подробное описывать другие методы не будем, полный исходный код можно будет посмотреть в приложении к данной работе. Так, описав основные возможности серверной части, перейдем к клиентской стороне приложения. Точкой входа в приложение является файл app.js. Он содержит в себе подключение модулей, необходимых для запуска Angular приложения. На листинге 3.6 показано подключение модулей.

Листинг 3.6. Подключение модулей в app.js

```
(function(){
  angular
     .module('QuadTreeCluster',[
        'ngMaterial',
        'geolocation',
        'gservice'
     ]);
})();
```

Исполняемый код оборачивается в самовызывающуюся функцию, которая служит для разграничения области видимости переменных и предотвращает возможные конфликты имен в будущем. Мы подключаем несколько модулей: ngMaterial – библиотека готовых визуальных компонентов; geolocation – служит для определения местонахождения клиента; gservice – служит для работы с Google картами.

Далее рассмотрим контроллер AddObjCtrl.js в котором реализовано добавление объекта на карту. Контроллер является связующим звеном в шаблоне проектирования MVC между представлением и моделью.

Рассмотрим метод add(), который добавляет объект на карту. На листинге 3.7 приведен код метода add().

Листинг 3.7. Описание метода add()

```
function add() {
  vm.formData.lat = parseFloat(vm.formData.lat);
  vm.formData.lon= parseFloat(vm.formData.lon);
  var objData = {
     title: vm.formData.title,
     category: vm.formData.category,
     location: [vm.formData.lon, vm.formData.lat]
  };
  // Saves the object data to the db
  $http.post('/places', objData)
     .then(function (response) {
       vm.formData.title = ‘’;
        vm.formData.category = ‘’;
        gservice.refresh(vm.formData.lat, vm.formData.lon);
        console.log('Object added successfully');
     },
     function(data){
        console.log('Error: ' + data);
     });
}
```

Рассмотрим подробнее этот метод. Вначале мы формируем объект, данные которого считываем из формы, которую заполнил пользователь. Далее выполняется запрос по пути places методом POST и передаем в качестве аргумента сформированный объект, затем возвращается promise, при успешном завершении которого выполняется вызов метода refresh() модуля

gservice, который обновляет массив маркеров на карте. При неудачном завершении promise пишется сообщение об ошибке в консоль.

Полную схему приложения можно увидеть на рис. 3.2

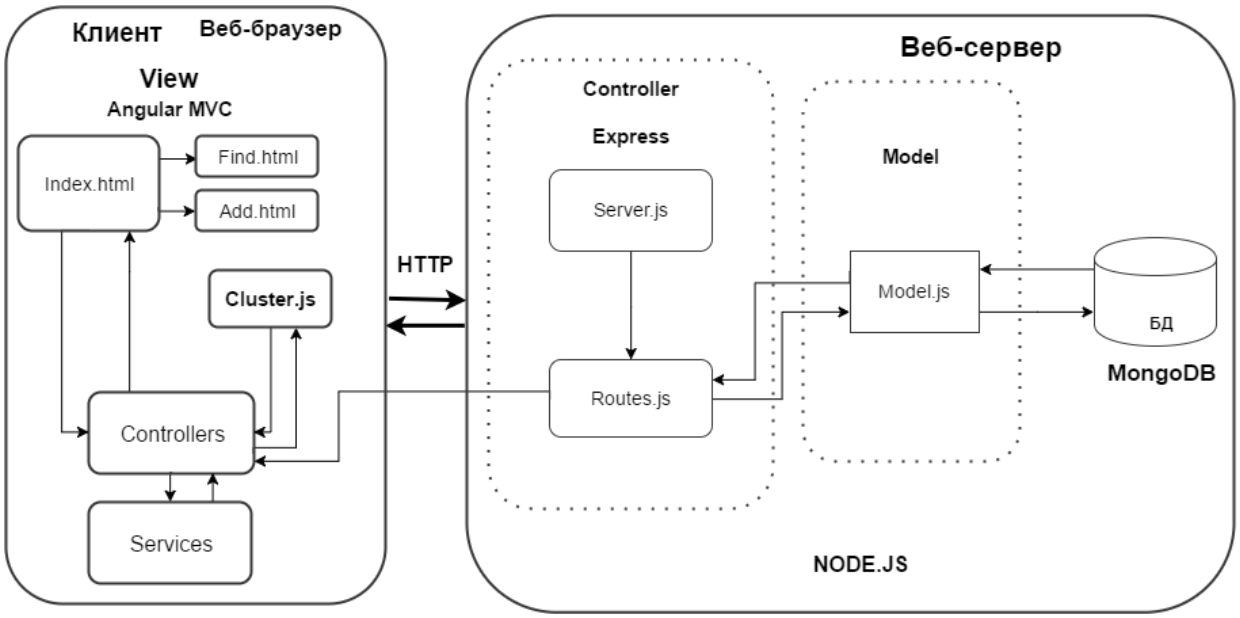


Рис. 3.2. Структурная схема приложения

Рассмотрим пользовательский интерфейс приложения на рис. 3.3

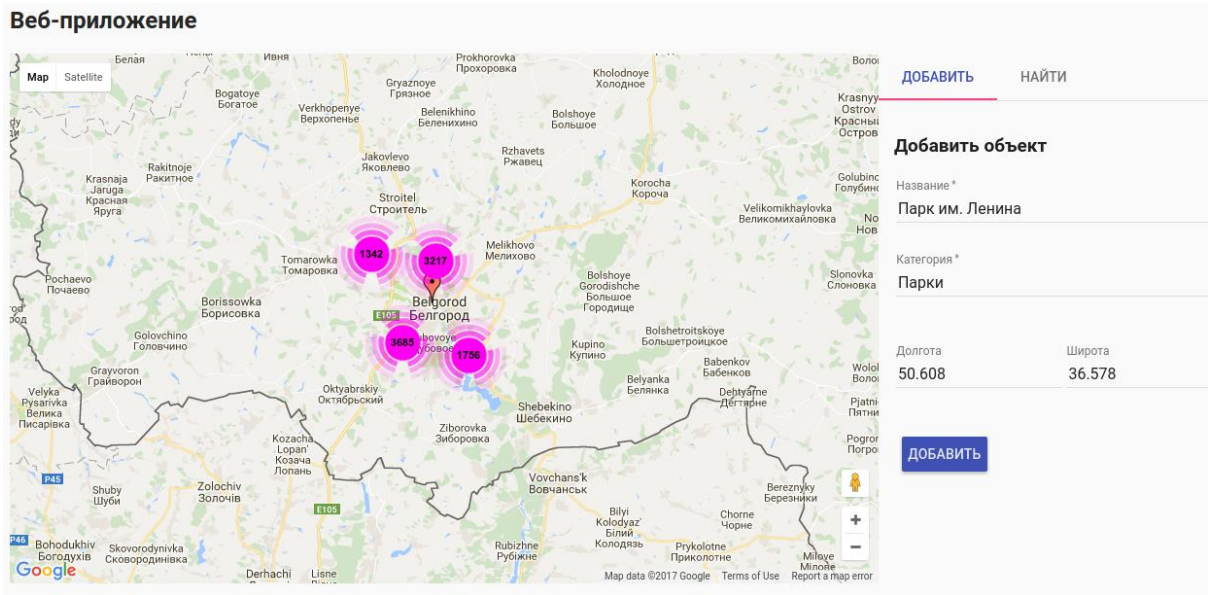


Рис. 3.3. Пользовательский интерфейс веб-приложения

На главном экране располагается географическая карта со множеством маркеров на ней. На данном скриншоте можем заметить, что точки на карте объединены в кластеры. В этом примере мы можем наблюдать 10 000 точек,

которые были обработаны алгоритмом. Справа от карты мы можем наблюдать форму добавления объекта. В ней представлено несколько полей ввода, такие как «название», «категория», долгота и широта, которые задаются кликом курсора на карте.

На второй вкладке по соседству находится форма поиска объектов по полю «категория». Пользователь выбирает категорию объектов и ему показываются только объекты, принадлежащие этой категории. Выборка происходит в базе данных и отдается пользователю без перезагрузки страницы. Алгоритм на клиентской стороне обрабатывает эти точки и формирует из них кластеры, исходя из уровня приближения (зума). Чем ближе зум, тем в более меньшие кластеры объединяются объекты. Алгоритм на лету обсчитывает все точки и формирует результат.

ГЛАВА 4 ИНТЕГРАЦИЯ И ТЕСТИРОВАНИЕ ПО

4.1 Интеграция ПО в геоинформационную систему «Активист»

Геоинформационная система «Активист» представляет собой комплекс программного обеспечения с клиент-серверной архитектурой. Задачами системы являются удобная для граждан консолидация сообщений Ленинградской области и управление жизненным циклом этих сообщений органами исполнительной власти. По замыслу заказчика, на главной панели веб-приложения системы должна располагаться географическая карта с нанесенными на ней маркерами, обозначающая сообщения пользователей о какой-либо проблеме или предложении. Эти сообщения разделяются по категориям, по которым нужно производить фильтрацию (выборку) сообщений, показанных на карте.

Так, существует два типа обращения: жалоба и предложение. Пользователь, зарегистрированный в системе, может подать до десяти обращений в день.

Одной из задач технического задания было задачей оптимального отображения большого количества объектов на карте без потери производительности и проблем восприятия пользователем информации. Данная задача была взята в основу для интеграции разработанного алгоритма в систему.

В настоящее время система находится в активной стадии разработки, поэтому данные реальной базы данных отсутствуют. Вместо реальных данных была произведена генерация большого количества объектов на территории Ленинградской области и в городе Санкт-Петербурге. Все замеры производительности тестов происходили на ноутбуке с конфигурацией процессора Intel Core i5 1.7GHz, и оперативной памятью объемом 6GB DDR3

1600MHz. Использовался набор данных в 10000 объектов. Также стоит упомянуть что все замеры производились в современном браузере Google Chrome 57 и операционной системой Ubuntu 16.04.

На рис. 4.1 представлен интерфейс главного окна прототипа веб-приложения.

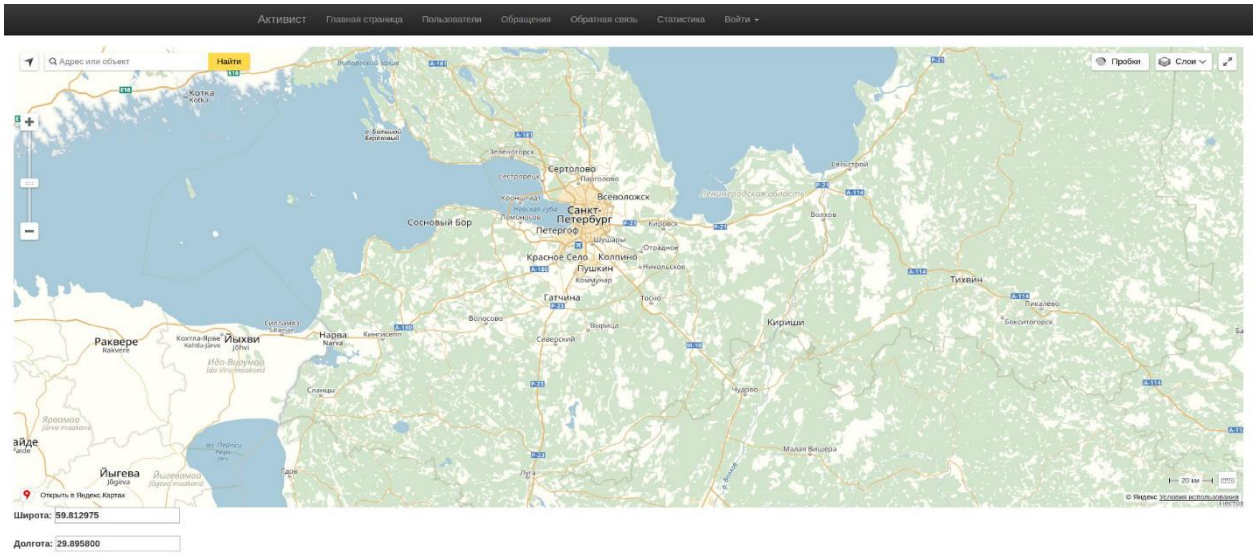


Рис. 4.1. Главное окно веб-приложения «Активист»

Далее следует рассмотреть структуру веб-приложения и определить возможность интеграции алгоритма в независимый модуль в клиентской части веб-приложения. На рис. 4.2 показана структура веб-приложения для работы с обращениями.

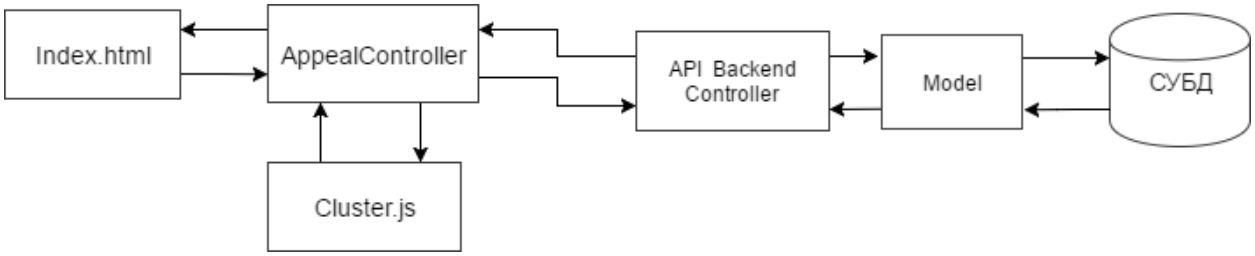


Рис. 4.2. Схема веб-приложения

Рассмотрим подробнее схему приложения. При запросе страницы с картой, в базе данных происходит выборка объектов. Далее она передается в модель, которая в свою очередь передает данные в контроллер на серверной

стороне, после этого данные передаются на клиентскую часть. Разработанный алгоритм мы поместили в модуль, называемый Cluster.js. Когда контроллер на клиентской стороне получает данные с сервера, он их предварительно обрабатывает, а именно передает их объектом JSON в модуль Cluster.js, который в свою очередь кластеризирует объекты и отдает обработанные данные обратно в контроллер. Последний формирует отображение объектов на странице пользователя.

Данный модуль был модифицирован под обработку данных, исходя из структуры объекта. Объект Appeal состоит из следующих поле: title, content, type, media, coords. Таким образом мы можем производить анализ объектов по полю type.

4.2 Тестирование алгоритма поиска и анализа свойств объектов на местности

После успешной интеграции модуля в систему, начнем тестирование алгоритма. Сгенерируем объекты на территории Ленинградской области в размере 10000 штук. Для наглядности выведем их на главную страницы, где расположена карта без кластеризации. На рис. 4.3 можно увидеть вывод снегерированных 10000 объектов на карту Ленинградской области.

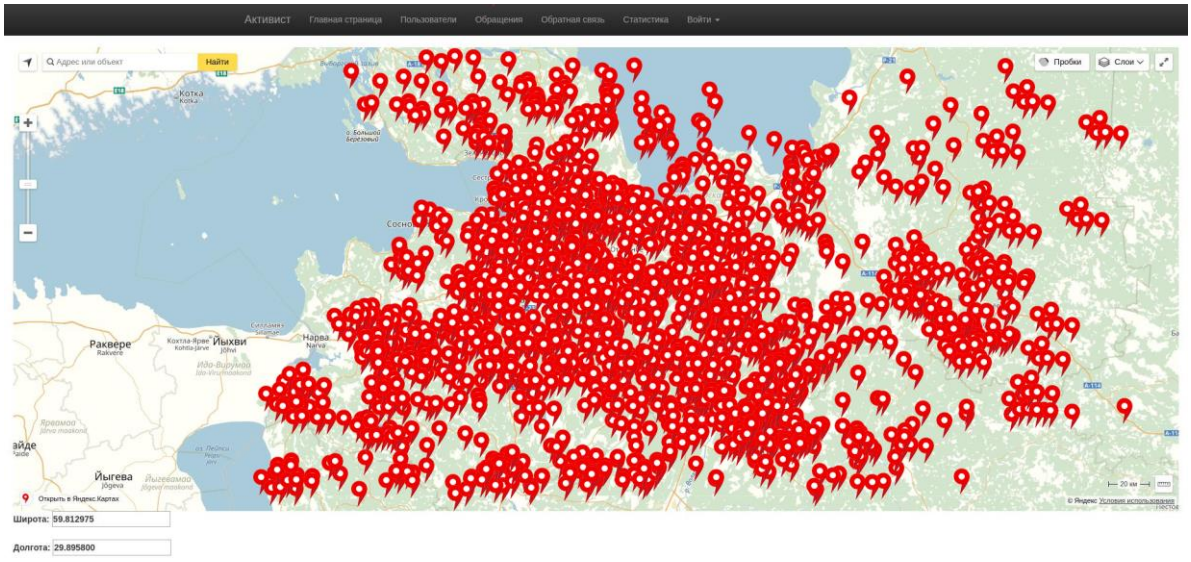


Рис. 4.3. Вывод сгенерированных данных

Как можно заметить, карта стала абсолютно нечитаемой из-за большого количества маркеров на карте. А страница в свою очередь зависла на несколько секунд, пока не вывелись абсолютно все маркеры. Таким образом, можно диагностировать проблему отображения большого количества объектов на карте. Ниже рассмотрим детально время загрузки, обработки и отображение маркеров на странице.

Сделаем замеры с помощью Chrome DevTools. Это инструмент, встроенный в браузер Google Chrome, позволяющий совершать отладку приложения, осуществлять анализ и аудит производительности веб-страницы, манипулировать с DOM-деревом, а также предлагает консоль разработчика. Мы будем использовать вкладку под названием Perfomance для замера производительности и потребления памяти приложением.

Проведем анализ того, что предлагает нам вкладка Perfomance. В самом верху так называемая панель timeline – позволяет наглядно посмотреть в какой момент началась работа скрипта, рендеринг и отрисовка страницы. Ниже расположена панель, где можно детально рассмотреть время выполнения той или иной функции и ее временной интервал. Чуть ниже панель для анализа потребления памяти. Нас интересует количество мегабайт в синем графике – это сколько занимают памяти созданные объекты. На последней панели показан график времени выполнения всех этапов загрузки страницы. Также Chrome DevTools предлагает возможности профилирования приложений – то есть детального анализа каждой вызванной функции и времени ее выполнения. Но в нашем случае эти возможности не понадобятся.

Стоит подробнее остановится на результатах профилирования веб-приложения. Так, загрузка данных составила 633мс, что является довольно медленным результатом. Работа скриптов составила более 10 секунд, а рендеринг занял больше одной секунды. Во время работы скриптов страница немного зависла, то есть перестала отвечать на какие-либо действия со стороны пользователя.

На рисунке 4.4 представлен скриншот вкладки при выводе 10000 объектов.

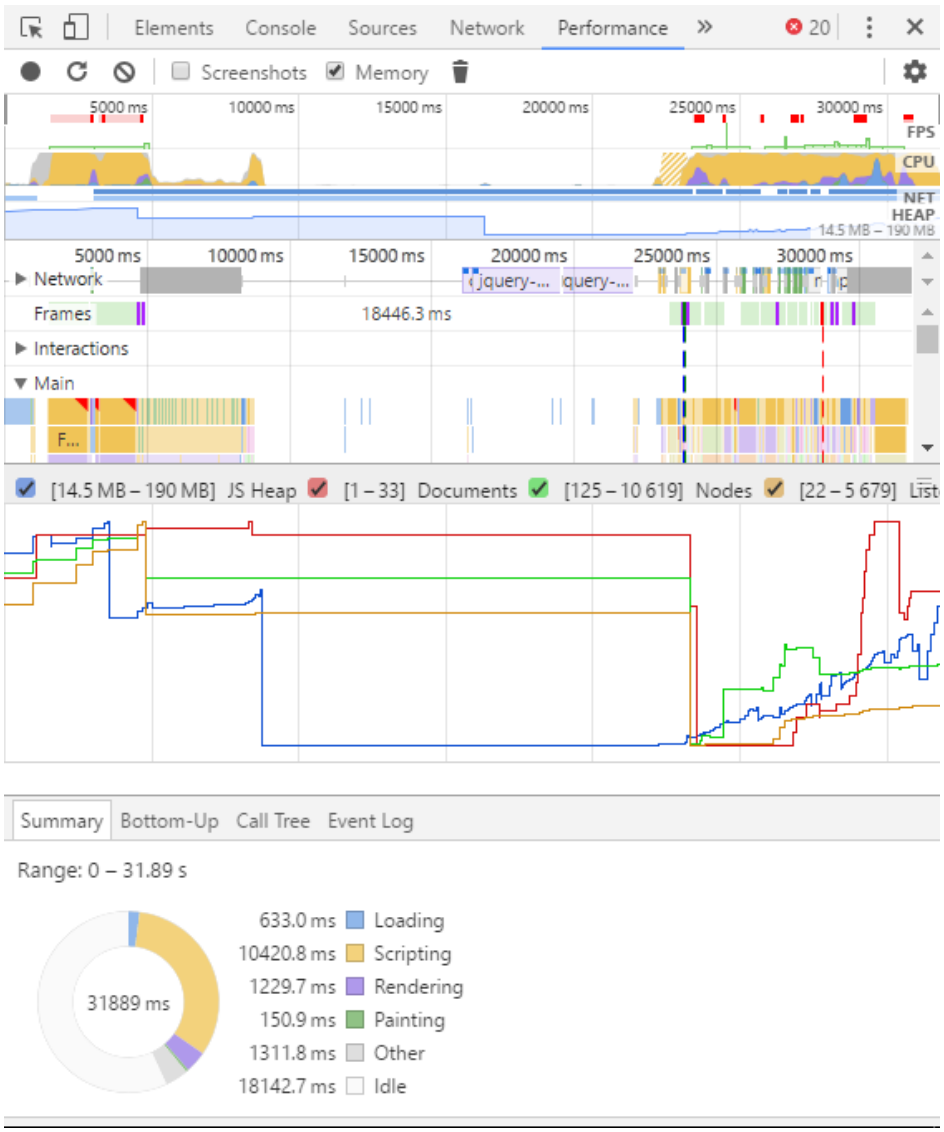


Рис. 4.4. Анализ производительности перед кластеризацией

Таким образом, можно с уверенностью сказать, что пользовательский опыт взаимодействия был негативный. Страница, которая долго грузится и работает медленно крайне плохо сказывается на следующих посещениях сервиса этим пользователем.

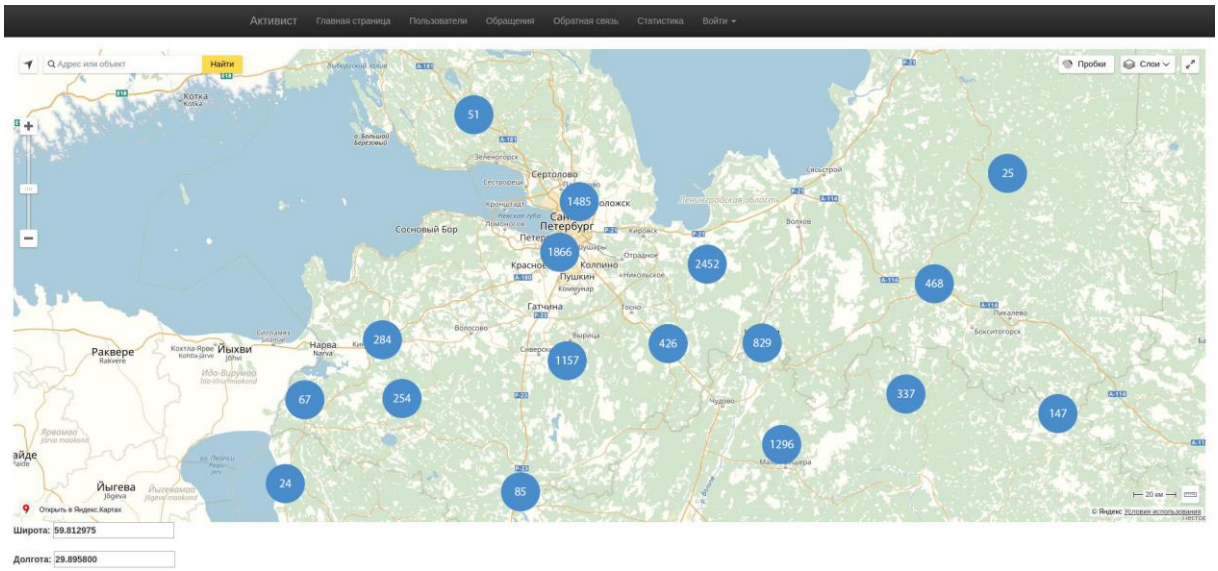


Рис. 4.5. Результат работы алгоритма кластеризации

После того, как мы интегрировали разработанный модуль в систему, покажем работу кластеризации алгоритма. На рисунке 4.5 можно увидеть результат работы алгоритма на тестовых данных.

Как можем заметить из графиков, представленных на рис. 4.6, загрузка данных составила всего 62 мс, а ее обработка всего 1,5 секунды, что показывает десятикратный прирост производительности на этапе первоначальной загрузки. Потребление памяти в этом случае многократно сократилось из-за отсутствия большого количества созданных объектов.

Подводя итог, можно сказать, что интеграция алгоритма в геоинформационную систему прошла успешно. Были показаны приемлемые результаты, отвечающие потребностям заказчика.

ЗАКЛЮЧЕНИЕ

В ходе выполнения выпускной квалификационной работы были спроектированы и разработаны веб-приложение и алгоритм поиска и анализа свойств объектов на местности. В результате чего были решены следующие задачи:

1. Исследованы существующие методы иерархического разбиения пространства и алгоритмы кластеризации данных.
2. Спроектированы база данных и веб-приложение.
3. Определены функциональные требования к алгоритму.
4. Разработано веб-приложение и реализован алгоритм.
5. Проведена интеграция и тестирование реализованного алгоритма.

В первой главе были рассмотрены основные методы иерархического разбиения пространства. Также был произведен обзор алгоритмов кластеризации данных и произведен выбор инструментальных средств разработки.

Во второй главе спроектировано веб-приложение, база данных для хранения объектов, определены функциональные требования к алгоритму.

В третье главе разработано веб-приложение, реализована база данных, и разработан алгоритм поиска и анализа свойств объектов на местности.

В четвертой главе была проведена интеграция алгоритма в геоинформационную систему «Активист». Проведено тестирование алгоритма и замеры производительности на разных объемах данных. Алгоритм показал заметный прирост в производительности приложения и улучшил пользовательский опыт взаимодействия с системой, о чем имеется акт апробации.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Воронцов К.В. Алгоритмы кластеризации и многомерного шкалирования. Курс лекций. МГУ, 2007. [Электронный ресурс]. – Режим доступа: http://www.ccas.ru/voron/download/Clustering.pdf, 01.05.2017
2. Мандель И. Д. Кластерный анализ. — М.: Финансы и Статистика, 1988.
3. Jain A., Murty M., Flynn P. Data Clustering: A Review. ACM Computing Surveys. 1999. Vol. 31, no. 3.
4. Дерево квадрантов [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/Дерево_квадрантов, 20.05.2017.
5. В. Б. Иванов, В. Н. Гиляров, А. А. Мусаев, “Пространственная кластеризация мест возникновения чрезвычайных ситуаций”, Тр. СПИИРАН, 24 (2013), 108–115.
6. Кластеризация маркеров [Электронный ресурс]. – Режим доступа: https://developers.google.com/maps/documentation/javascript/marker-clustering?hl=ru#markerclusterer, 07.05.2017.
7. Князь Д. Анализ основных алгоритмов кластеризации многомерных данных. – LAP Lambert Academic Publishing, 2014. – 64 с.
8. Бериков В. С., Лбов Г. С. Современные тенденции в кластерном анализе // Всероссийский конкурсный отбор обзорно-аналитических статей по приоритетному направлению «Информационно-телекоммуникационные системы», 2008. — 26 с.
9. Часовских А. Обзор алгоритмов кластеризации данных [Электронный ресурс]. – Режим доступа: https://habrahabr.ru/post/101338/, 02.05.2017.

10. Жамбю М. Иерархический кластер-анализ и соответствия. — М.: Финансы и статистика, 1988. — 345 с.

11. ObjectManager в API Яндекс.Карт [Электронный ресурс]. – Режим доступа: https://yandex.ru/blog/mapsapi/53158, 05.05.2017.

12. Документация MongoDB [Электронный ресурс]. – Режим доступа: https://docs.mongodb.com/manual/, 02.05.2017.

13. Документация Node.js [Электронный ресурс]. – Режим доступа: https://nodejs.org/dist/latest-v6.x/docs/api/, 29.04.2017.

14. Документация Angular.js [Электронный ресурс]. – Режим доступа: https://docs.angularjs.org/guide, 25.04.2017.

15. Флэнаган Д. JavaScript. Подробное руководство. – Пер. с англ. – СПб: Символ Плюс, 2008. – 992 с.

16. Michael McMillan Data structures and algorithms with JavaScript. – O'Reilly Media, Inc., 2014. – 246 с.

17. Э.Гамма, Р.Хелм, Р.Джонсон, Д.Влиссидес. Приемы объектно-ориентированного проектирования Паттерны проектирования - СПб. ПИТЕР, 2006 – 306 с.

18. Bereuter, Pia; Weibel, Robert (2012). Algorithms for on-the-fly generalization of point data using quadtrees [Электронный ресурс]. –Режим доступа: http://www.zora.uzh.ch/74663/1/2012_BereuterP_Bereuter_Weibel_AutoCarto2012.pdf, 24.04.2017.

ПРИЛОЖЕНИЕ 1

Исходный код файла server.js

```
var express        = require('express');
var mongoose       = require('mongoose');
var port           = process.env.PORT || 3000;
var morgan         = require('morgan');
var bodyParser     = require('body-parser');
var dummyjson      = require('dummy-json');
// var methodOverride  = require('method-override');
var app            = express();
var Place          = require('./app/model.js');
mongoose.connect("mongodb://localhost/QuadTreeCluster");
   // Logging and Parsing
app.use(express.static(__dirname + '/public'));                 // sets the static files location to public
app.use('/bower_components',  express.static(__dirname + '/bower_components'
app.use(morgan('dev'));                                         // log with Morgan
   app.use(bodyParser.json());                                  // parse application/json
   app.use(bodyParser.urlencoded({extended: true}));
   app.use(bodyParser.text());
   app.use(bodyParser.json({ type: 'application/vnd.api+json'}));
   require('./app/routes.js')(app);
   app.listen(port);
   console.log('App listening on port ' + port);
```

ПРИЛОЖЕНИЕ 2

Исходный код файла cluster.js

```
var quadtree = function(options)
{
   "use strict";

   var opts = {},
      tree,
      allItems = [];

   // constants for quadrants
   var TOP_LEFT = 'tl',
      TOP_RIGHT = 'tr',
      BOTTOM_LEFT = 'bl',
      BOTTOM_RIGHT = 'br';

   /**
    * Init function is called on tree instantiation and clearing.
    * It checks tree options and creates root node of the tree.
    * @param {Object} options
    */
   function init(options)
   {
      if (! ('top' in options) || ! ('left' in options) || ! ('bottom' in options) || ! ('right' in
options)) {
         throw new    Error('not enough init options');
      }
      opts = {
         top: options.top,
         left: options.left,
         bottom: options.bottom,
         right: options.right,
         max_per_cell: options.max_per_cell || 2,
         max_depth: options.max_depth || 4,
         debug_mode: options.debug_mode || false
      };

      tree = new Node(opts.top, opts.left, opts.bottom, opts.right, 0);
   }
```

```
    init(options);

    /**
     * Main function for creating index nodes
     * @constructor
     * @param {number} top
     * @param {number} left
     * @param {number} bottom
     * @param {number} right
     * @param {number} depth - 0 for root node, every other node has depth as
parent's + 1
     */
    function Node(top, left, bottom, right, depth)
    {
        this.top = top;
        this.left = left;
        this.bottom = bottom;
        this.right = right;
        this.depth = depth;

        this.items = [];
        this.nodes = undefined;

        /**
         * Checks if node can have point with such coordinates
         * @param {number} x
         * @param {number} y
         * @returns {boolean}
         */
        this.containsPoint = function containsPoint(x, y)
        {
            return pointIsInBounds(x, y, this.top, this.left, this.bottom, this.right);
        };

        /**
         * Checks  if node can have some points from a given range
         * @param {number} top
         * @param {number} left
         * @param {number} bottom
         * @param {number} right
         * @returns {boolean}
         */
        this.containsRangePart = function containsRangePart(top, left, bottom, right)
        {
```

```
        return rangesIntersect(top, left, bottom, right, this.top, this.left, this.bottom,
this.right);
    };

    /**
     * It's used to insert yet another point into this node.
     * It checks if node has to contain it itself or delegate to child nodes.
     * @param {{x: (number), y: (number), i: (number)}} itemForIndex
     * @returns {boolean}
     */
    this.insert = function insert(itemForIndex)
    {

        // if this point is out of bounds, do nothing
        if (! this.containsPoint(itemForIndex.x, itemForIndex.y)) {
            return false;
        }
        // if we have nodes, delegate
        if (this.nodes)
        {
            var nodes = this.nodes;
            if (nodes[TOP_LEFT].insert(itemForIndex)) return true;
            if (nodes[TOP_RIGHT].insert(itemForIndex)) return true;
            if (nodes[BOTTOM_LEFT].insert(itemForIndex)) return true;
            if (nodes[BOTTOM_RIGHT].insert(itemForIndex)) return true;

            if (opts.debug_mode) {
                throw new Error('item was not inserted into any node: ' +
itemAsString(itemForIndex));
            }
            return false;
        }
        // if this is a leaf node and we can add new items, add
        if (this.items.length < opts.max_per_cell || this.depth >= opts.max_depth)
        {
            this.items.push(itemForIndex);
            return true;
        }
        // otherwise, divide node and insert item again
        this.divide();
        return this.insert(itemForIndex);
    };

    /**
```

```
        * If there's not enough space for another point in current node, it's becoming
a parent of 4 child nodes
        * and all it's items are moved to this nodes
        * @returns {boolean}
        */
       this.divide = function divide()
       {
          var nodes = this.nodes = {},
             halfWidth = (this.right - this.left) / 2,
             halfHeight = (this.bottom - this.top) / 2,
             debug = opts.debug_mode,
             resInsert;
          nodes[TOP_LEFT] = new Node(this.top, this.left, this.top + halfHeight,
this.left + halfWidth, depth + 1);
          nodes[TOP_RIGHT] = new Node(this.top, this.left + halfWidth, this.top +
halfHeight, this.right, depth + 1);
          nodes[BOTTOM_LEFT] = new Node(this.top + halfHeight, this.left,
this.bottom, this.left + halfWidth, depth + 1);
          nodes[BOTTOM_RIGHT] = new Node(this.top + halfHeight, this.left +
halfWidth, this.bottom, this.right, depth + 1);

          for (var i = 0, items = this.items, l = items.length; i < l; i ++) {
             resInsert = this.insert(items[i]);
             if (debug && ! resInsert) {
                throw new Error('Item not inserted while dividing: ' +
itemAsString(items[i]));
             }
          }
          this.items = [];
          return true;
       };

       /**
        * It's for retrieving data from index.
        * It's duplicates aware.
        * @param {number} top
        * @param {number} left
        * @param {number} bottom
        * @param {number} right
        * @param {Object} storeValidIndices - that is where valid index data will be
stored to.
        */
       this.queryRange = function queryRange(top, left, bottom, right,
storeValidIndices)
```

```
        {
            // if out of bounds, do nothing
            if (! this.containsRangePart(top, left, bottom, right)) {
                return;
            }
            // if node has child nodes, delegate
            if (this.nodes)
            {
                var nodes = this.nodes;
                nodes[TOP_LEFT].queryRange(top, left, bottom, right,
storeValidIndices);
                nodes[TOP_RIGHT].queryRange(top, left, bottom, right,
storeValidIndices);
                nodes[BOTTOM_LEFT].queryRange(top, left, bottom, right,
storeValidIndices);
                nodes[BOTTOM_RIGHT].queryRange(top, left, bottom, right,
storeValidIndices);
            }
            // otherwise node has items, test each of them and push valid ones into given
storage object
            if (this.items.length)
            {
                for (var i = 0, items = this.items, l = items.length, item; i < l; i ++)
                {
                    item = items[i];
                    if (pointIsInBounds(item.x, item.y, top, left, bottom, right)) {
                        storeValidIndices[item.i] = true;
                    }
                }
            }
            // if no items in leaf node, do nothing
        };
    }

    //-------- helpers

    /**
     * Checks if point's coordinates are in specified range.
     * It includes all edges, because we're filtering out duplicates later.
     * @param {number} x - point's coordinate
     * @param {number} y - point's coordinate
     * @param {number} top - range's coordinate
     * @param {number} left - range's coordinate
     * @param {number} bottom - range's coordinate
```

```
     * @param {number} right - range's coordinate
     * @returns {boolean}
     */
    function pointIsInBounds(x, y, top, left, bottom, right)
    {
        if (x >= left && x <= right && y >= top && y <= bottom) {
            return true;
        }
        return false;
    }

    /**
     * Checks if range's intersects with specified range.
     * It includes all edges, because we're filtering out duplicates later.
     * @param {number} top
     * @param {number} left
     * @param {number} bottom
     * @param {number} right
     * @param {number} topRange
     * @param {number} leftRange
     * @param {number} bottomRange
     * @param {number} rightRange
     * @returns {boolean}
     */
    function rangesIntersect(top, left, bottom, right, topRange, leftRange,
bottomRange, rightRange)
    {
        if (left <= rightRange && right >= leftRange && top <= bottomRange &&
bottom >= topRange) {
            return true;
        }
        return false;
    }

    function isArray(item)
    {
        return Object.prototype.toString.apply(item) === '[object Array]';
    }

    function isNumber(num)
    {
        return typeof num === 'number';
    }
```

```
    /**
     * It's for debugging purposes
     * @param {Object} item
     * @returns {string}
     */
    function itemAsString(item)
    {
        if (! item) {
            return 'empty item';
        }
        return '{x: ' + item.x + ', y: ' + item.y + '}';
    }

    /**
     * It's used to sort by closeness
     * @param {{x: (number), y: (number)}} p1 - point 1
     * @param {{x: (number), y: (number)}} p2 - point 2
     * @returns {number} - distance between points
     */
    function distancePow2(p1, p2)
    {
        return Math.pow(p1.x - p2.x, 2) + Math.pow(p1.y - p2.y, 2);
    }

    /**
     * This function gets all real data by index data's ids.
     * @param {Object} index - index data stored by real data's array id
     * @param {{x: (number), y: (number)}} sortClosestTo - sorts all results by
closeness to this coordinates
     * @returns {Array}
     */
    function itemsByIndex(index, sortClosestTo)
    {
        var items = [];
        for (var i in index)
        {
            if (index.hasOwnProperty(i)) {
                items.push(allItems[i]);
            }
        }

        if (sortClosestTo && sortClosestTo.x !== undefined && sortClosestTo.y !==
undefined) {
            items.sort(function (a, b) {
```

```
            return distancePow2(a, sortClosestTo) < distancePow2(b, sortClosestTo)
? -1 : 1;
        });
    }

    return items;
  }

  /**
   * This item's representation will be put into the index tree
   * @param {Object} item
   * @param {number} index
   * @returns {{x: (number), y: (number), i: (number)}}
   */
  function itemToIndex(item, index)
  {
    return {x: item.x, y: item.y, i: index};
  }

  /**
   * Checks if item can be stored into the tree
   * @param item
   * @returns {boolean}
   */
  function checkIsValid(item)
  {
    if (! ('x' in item) || ! ('y' in item))
    {
      if (opts.debug_mode) {
        throw new Error('invalid item ' + (item && itemAsString(item)));
      }
      return false;
    }
    return true;
  }

  /**
   * Public methods to work with indexed quadtree
   */
  return {
    /**
     * Inserts items with coordinattes into the index tree
     * @param {Object|Array} item - it could be either object with params
defining point or array of that objects
```

```
     *
     * point params are:
     * {
                *     x: center x coordinate,
                *     y: center x coordinate
                * }
     */
    insert: function _insert(item)
    {
       if (! tree || (! 'top' in opts)) {
          throw new Error('Quadtree is not initialized');
       }

       var debug = opts.debug_mode,
          res;

       if (isArray(item))
       {
          var totalLengthBefore = allItems.length,
             l = item.length,
             cur;
          if (l > 0 && checkIsValid(item[0]))
          {
             for (var i = 0; i < l; i ++)
             {
                cur = item[i];
                allItems.push(cur);
                res = tree.insert(itemToIndex(cur, totalLengthBefore + i));
                if (debug && ! res) {
                   throw new Error('Item not inserted (' + i + '): ' +
itemAsString(cur));
                }
             }
          }
       }
       else
       {
          if (checkIsValid(item))
          {
             allItems.push(item);
             res = tree.insert(itemToIndex(item, allItems.length - 1));
             if (debug && ! res) {
                throw new Error('Item not inserted: ' + itemAsString(item));
             }
```

```
            }
        }
    },
    /**
     * Get all points in a range with coordinates
     * @param {number} top
     * @param {number} left
     * @param {number} bottom
     * @param {number} right
     * @param {boolean} sortByCloseness
     * @returns {Array}
     */
    queryRange: function _queryRange(top, left, bottom, right, sortByCloseness)
    {
        if (! isNumber(top) || ! isNumber(left) || ! isNumber(bottom) || !
isNumber(right))
        {
            if (opts.debug_mode) {
                throw new Error('Invalid arguments for queryRange');
            }
            return [];
        }
        var index = {};
        tree.queryRange(top, left, bottom, right, index);
        return itemsByIndex(index, sortByCloseness && {x: (left + right) / 2, y:
(top + bottom) / 2});
    },
    /**
     * Get all points in a range by center coordinates, width and height
     * @param {number} x - center x
     * @param {number} y - center y
     * @param {number} w - width
     * @param {number} h - height
     * @param {boolean} sortByCloseness
     * @returns {Array}
     */
    queryRangeByCenter: function _queryRangeByCenter(x, y, w, h,
sortByCloseness)
    {
        if (! isNumber(x) || ! isNumber(y) || ! isNumber(w) || ! isNumber(h))
        {
            if (opts.debug_mode) {
                throw new Error('Invalid arguments for queryRangeCenter');
            }
```

```
            return [];
        }
        var index = {};
        tree.queryRange(y - h / 2, x - w / 2, y + h / 2, x + w / 2, index);
        return itemsByIndex(index, sortByCloseness && {x: x, y: y});
    },
    /**
     * Get all points in a range represented by object with center coordinates, width and height
     * @param {{x: (number), y: (number), w: (number), h: (number)}} obj
     * @param {boolean} sortByCloseness
     * @returns {Array}
     */
    queryContainedBy: function _queryContainedBy(obj, sortByCloseness)
    {
        if (! obj || ! isNumber(obj.x) || ! isNumber(obj.y) || ! isNumber(obj.w) || ! isNumber(obj.h))
        {
            if (opts.debug_mode) {
                throw new Error('Invalid arguments for queryContainedBy');
            }
            return [];
        }
        var index = {};
        tree.queryRange(obj.y - obj.h / 2, obj.x - obj.w / 2, obj.y + obj.h / 2, obj.x + obj.w / 2, index);
        return itemsByIndex(index, sortByCloseness && {x: obj.x, y: obj.y});
    },
    /**
     * Clears index tree
     */
    clear: function _clear()
    {
        tree.items = [];
        tree.nodes = undefined;
        init(opts);
    }
  }
};
```