УДК: 004.652
DOI:

А.А. Михайлов, А.С. Васильев, Н.М. Селивёрстов

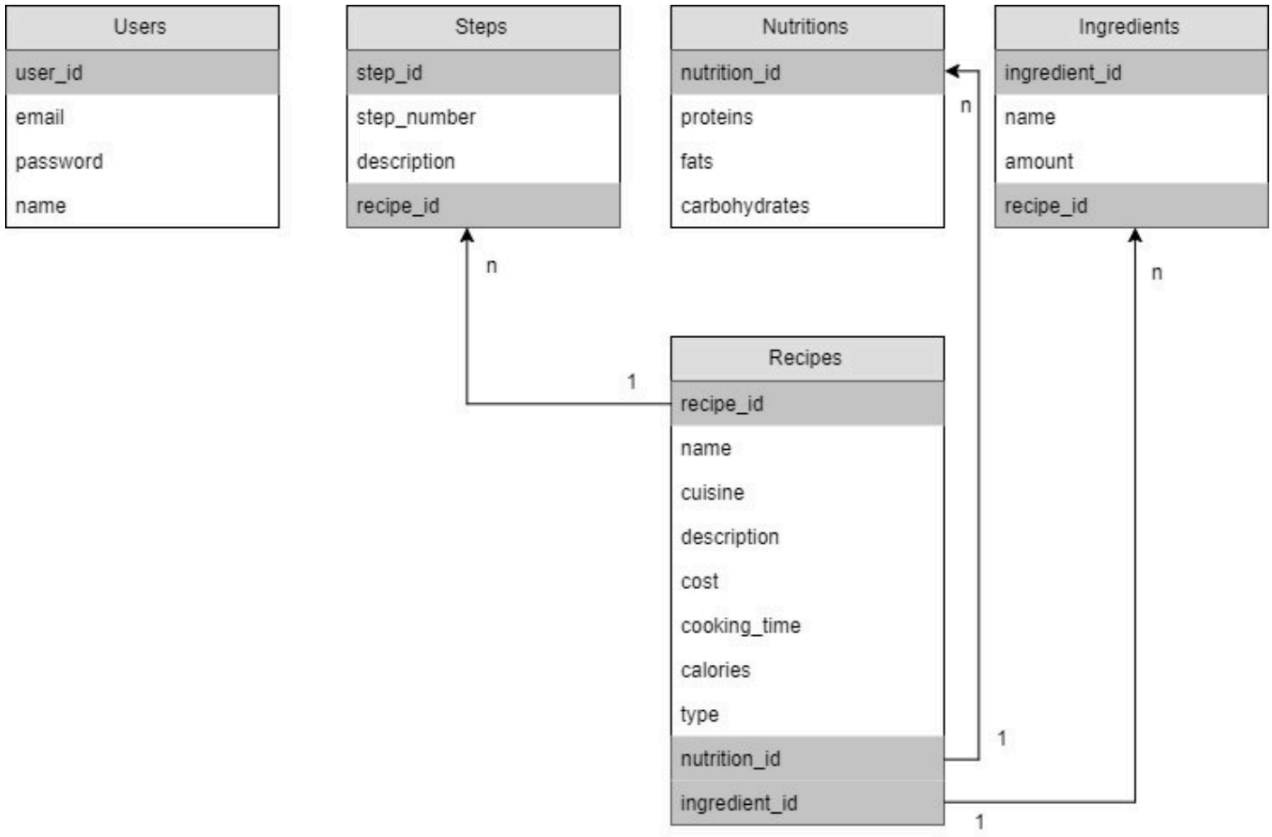
АНАЛИЗ РЕЛЯЦИОННОЙ И НЕРЕЛЯЦИОННОЙ МОДЕЛИ ДАННЫХ ПРИ РАЗРАБОТКЕ ENTERPRISE-ПРИЛОЖЕНИИ

Различие в подходе к хранению и представлению данных порождает актуальность выбора используемых технологий на этапе проектирования приложения, также внутреннее устройство различных систем управления базами данных влияет на особенности работы с ними. *В рамках статьи будет спроектирована структура данных для конкретной рассматриваемой в статье предметной области. Результатом данной статьи будет вывод о рентабельности и оптимальности использования различных моделей данных. Так же будет произведено сравнение существующих реализаций доступа к данным.*

Ключевые слова: базы данных, структуры данных, модель данных, реляционные, нереляционные, проектирование баз данных

В рамках данной научной статьи было проведено исследование реляционных и нереляционных моделей данных на примере следующей предметной области: разработке enterprise - приложения для планирования недельного рациона с возможностью установки ограничений бюджету и калорийности. При реализации в качестве СУБД была выбрана MongoDB.

Модель данных



Данная модель представляет из себя ER-диаграмму. На ее основе можно проектировать как нереляционную БД, так и реляционную. В нашем случае используется популярная нереляционная документо-ориентированная СУБД - MongoDB. Суть документо-ориентированного подхода заключается в том, что у нас имеются некоторые коллекции, которые представлены в машиночитаемом виде (зачастую JSON) и могут содержать в себе любую структуру данных. Задача приложения - доставать нужные данные(рецепты) используя некоторые параметры. В нашей модели главная сущность - это рецепт, также отсутствуют сложные связи и отношения. Поэтому в нашем случае довольно легко и выгодно прибегнуть к стратегии "встраивания" - все (или некоторые) зависимые сущности внедряются в главную сущность - рецепт. В результате получается примерно такая структура, с которой в дальнейшем легко будет работать:

```
{
   _id: RECIPE_ID,
```

```
    name: "NAME",
    cuisine: "CUISINE",
    description: "DESCRIPTION",
    cost: "COST",
    cooking_time: "COOKING TIME",
    calories: "CALORIES",
    type: "TYPE",
    nutrition: [{
            proteins: "PROTEINS",
            fats: "FATS",
            carbohydrates: "CARBOHYDRATES"
    }],
    ingredients: [{
            name: "NAME",
            amount: "AMOUNT",
            units: "UNITS"
    }, ...],
    steps: [{
            step_number: "STEP NUMBER",
            description: "DESCRIPTION"
    }, ...]
}
```

Отдельная коллекция пользователей предварительно будет иметь следующий вид:

```
{
    _id: USER_ID,
    name: "NAME",
    email: "EMAIL",
    password: "PASSWORD"
}
```

Пример уже существующих данных в MongoDB:

```
{
    "_id" : ObjectId("5a036851e972795b09ec49de"),
    "name" : "Плов",
    "cuisine" : "Узбекистан",
    "description" : "Лучшая пища на планете. Еда богов. Пловец от узбеков с говядиной и вы захотите жить еще
больше.",
    "cost" : 400,
    "cookingTime" : "25 минут",
    "calories" : 500,
    "type" : "ужин",
    "nutrition" : [
        {
            "proteins" : 65,
            "fats" : 78,
            "carbohydrates" : 120
        }
    ],
    "ingredients" : [
        {
            "name" : "Рис",
            "amount" : 500,
            "units" : "грамм"
```

```
      },
      {
        "name" : "Лук",
        "amount" : 2,
        "units" : "штуки"
      }
    ],
    "steps" : [
      {
        "step_number" : 1,
        "description" : "Засыпать риса."
      },
      {
        "step_number" : 2,
        "description" : "Добавить лука."
      },
      {
        "step_number" : 3,
        "description" : "Поставить варить на 20 минут."
      }
    ]
}
```

Расчет затрат памяти

Ниже будут указаны средние значения для полей, они актуальны для mongoDB, для реляционной базы данных придется указывать фиксированные значения(максимальные). ###Fields: name - type String, average length - 16 символов. cuisine - type String, average length - 10 символов. description - type String, average length - 240 символов. cost - type Int. cookingTime - 8 символов. calories - type Int. type - type String, average length - 5 символов. nutrition - 3 Int (proteins, fats, carbohydrates). ingredients - 1 Int (amount) and 2 String (name - average length 8 символов, units - average length 5 символов). steps - 1 Int (step_number) and 1 String (description - average length 240 символов).

Реляционная база данных

Подсчитаем затраты памяти для реляционной модели. У нас используются данные двух типов - String и Int. Вычислять объемы будем пока что с использованием абстрактных String и Int:

```
Таким образом, одна запись в таблице User будет занимать 1 Int [4 байта] и 3 String [20 + 20 + 64 = 104 байта].
Одна запись в таблице Steps будет занимать 2 Int и 1 String (408 байт), соответственно.
Nutrition = 4 Int [16 байт].
Ingredients = 1 Int [4 байта] + 2 String [20 + 40 = 60 байт].
Recipes = 3 Int + 5 String + Steps * n1 + Nutrition * 1 + Ingredients * n2 =>
Recipes = 3 Int [12 байт] + 5 String [50 + 20 + 500 + 20 + 10 = 600 байт] + (2 Int + 1 String [8 + 400 = 408 байт]) * n1 + 4 Int [16 байт] + (2 Int [8 байт] + 1 String [400 байт]) * n2.
Предположим, что среднее число Steps для каждого рецепта будет равно 5, а среднее число Ingredients - 5.
Тогда суммарный объем памяти во всех таблицах для одного рецепта:
3 Int [12 байт] + 5 String [600 байт] + (2 Int + 1 String) [408 байт] * 5 + 4 Int [16 байт] + (2 Int + 1 String) [68 байт] * 5
```

= 3 Int [12 байт] + 5 String [600 байт] + 10 Int [40 байт] + 5 String [408 * 5 = 2040 байт] + 4 Int [16 байт] + 10 Int [40 байт] + 5 String [5 * 60 = 300 байт]
= 27 Int [108 байт] + 15 String [600 + 2040 + 300 = 2940 байт]
= 2940 байт на одну запись рецепта в таблице и 108 байт на одну запись пользователя в таблице.

Нереляционная база данных

Подсчитаем затраты памяти для нереляционной модели. У нас также используются данные двух типов - String и Int.

Таким образом, одна запись в коллекции User хранит 1 Int [4 байта] и 3 String [7 + 7 + 64 = 78 байт].
Одна запись в таблице Steps будет занимать 1 Int и 1 String (244 байта), соответственно.
Nutrition = 4 Int [16 байт].
Ingredients = 1 Int [4 байта] + 2 String [8 + 5 = 13 байт].
Recipes = 2 Int + 5 String + Steps * n1 + Nutrition * 1 + Ingredients * n2 =>
Recipes = 2 Int [8 байт] + 5 String [16 + 10 + 240 + 8 + 5 = 279 байт] + (1 Int [4 байта] + 1 String [240 байт]) [4 + 240 = 244 байта] * n1 + 4 Int [16 байт] + (1 Int [4 байт] + 2 String [8 + 5 = 13 байт]) [4 + 13 = 17 байт] * n2.
Предположим, что среднее число Steps для каждого рецепта будет равно 5, а среднее число Ingredients - 5.
Тогда суммарный объем памяти во всех таблицах для одного рецепта:
2 Int [8 байт] + 5 String [279 байт] + (1 Int + 1 String) [244 байт] * 5 + 4 Int [16 байт] + (2 Int + 1 String) [17 байт] * 5
= 2 Int [8 байт] + 5 String [279 байт] + 10 Int [40 байт] + 5 String [244 * 5 = 1220 байт] + 4 Int [16 байт] + 10 Int [40 байт] + 5 String [5 * 17 = 85 байт]
= 27 Int [108 байт] + 15 String [279 + 1220 + 85 = 1584 байт]
= 1584 байт на одну запись рецепта в таблице и 78 байт на одну запись пользователя в таблице.

Итого, объемы нереляционной базы данных для рецептов будут занимать в 1.85 раз меньше памяти, а для списка пользователей - в 1.4 раза меньше. насколько это критично, нужно решать каждому разработчику отдельно. Но с уверенностью можно сделать вывод о том, что в нашем случае нереляционная модели будут занимать меньшие объемы памяти. Однако стоит лишь немного усложнить задачу и добавить возможность архивирования истории меню для каждого пользователя, то мы сразу же сталкиваемся с двумя проблемами в нереляционной модели - абсолютно полное дублирование данных о рецептах (расчеты будут приведены ниже), либо встраивание в историю id рецепта (тогда придется делать множество запросов, чтобы получать каждый отдельный рецепт отдельно), плюс для каждого изменения оригинального рецепта нужно делать отдельные запросы, чтобы изменять данные о рецептах в архиве. В реляционной же модели таких проблем не предвидится, так как записи из архива будут ссылаться на оригинальные записи в таблице Recipes, благодаря этому решается и вторая проблема (хотя это спорный вопрос по поводу того, проблема ли это).

Ни для кого не секрет, что дублирование данных в нереляционных СУБД - это абсолютно нормальная практика. Однако за счёт этого появляется возможность упростить запросы и время доступа к любым данным, об этом будет написано немного ниже.

Допустим пользователь решает сохранить одно из его недельных меню. Тогда мы можем наблюдать следующую картину:
недельное меню - это 7 дней, каждый день по три приема пищи (завтрак, обед и ужин), таким образом это 21 рецепт лишь для одного меню у одного юзера. Объемы памяти для данных в архиве:
1 Int [4 бата]- id пользователя, к примеру.
1 String [6 сивмолов = 6 байт] - дата составления меню.
1 Int [4 байта] - ограничения по калориям.
1 Int [4 байта]- ограничения по бюджету.

```
1584 байта * 21 - все хранимые рецепты.
-----------------------------------------------
Итого: (3 Int [12 байт] + 1 String [6 байт] + 1584 байта) * 21 =
= 3 Int [12 байт] + 1 String [6 байт] + 33264 байт
= 12 + 6 + 33264 = 33282 байта.

В том время, как в релиционной модели затраты будут 3 Int и 1 String.
+ ссылки на записи рецептов в таблице Recipes. А это к слову 4 Int [16 байт] + 1 String [8 байт] = 24 байта.
Выводы очевидны.
```

Оценка и пример запросов к БД

Нереляционная модель

В нашем случае используется такой механизм нереляционной базы данных MongoDB, как агрегация. Мы просто указываем параметры для поиска и находим нужные нам рецепты, абсолютно ничего сложного и интуитивно понятно:

```
List<AggregationOperation> operations = new ArrayList<>();
operations.add(new MatchOperation(Criteria.where("calories").lte(caloriesUpperBound).gte(caloriesLowBound)));
operations.add(new MatchOperation(Criteria.where("cost").lte(budgetUpperBound).gte(budgetLowBound)));
operations.add(new SortOperation(new Sort(sortDirection, "calories")));
operations.add(new MatchOperation(Criteria.where("type").is(type)));
operations.add(new LimitOperation(7));
Aggregation agg = newAggregation(operations);
List<Recipe> recipes = mongoTemplate.aggregate(agg, Recipe.class, Recipe.class).getMappedResults();
```

Это небольшой участок кода программы, где и происходит запрос к БД с целью получить все необходимые данные по нужным критериям. То есть наша задача - это составить список агрегационных операций и передать их в метод aggregate объекта mongoTemplate. Из списка операций видно, что мы запрашиваем рецепты, у который калорийность меньше верхнего порога и больше нижнего порога. Точно также мы поступаем и с ценой блюда, а также указываем тип рецепта - завтрак, обед или ужин, и устанавливаем лимит в 7 элементов (потому что в данном случае мы составляем меню на неделю и нам необходимо, к примеру, 7 рецептов для завтраков). Предположим, что у нас будет использоваться такая функциональность, как архив. Как было описано выше, мы может поступить в данном случае двумя способами:

1) В элементах коллекции архив содержать ссылки(id) на все необходимые рецепты для каждого сохраненного меню. В таком случае мы избегаем большого количества дублирования данных (расчеты можно найти выше).

2) В элементах коллекции хранить не ссылки, а полноценные сущности recipe, таким образом.

Давайте проанализируем оба подхода. Предположим, что история у нас будет выводиться по 5 меню на странице. Таким образом нужно будет сделать один запрос для поиска нужных архивов какого-то пользователя и потом для найденных 5 архивированных меню сделать по 21 запросу на каждый рецепт - 21 * 5 = 105 запросов. Итого, 106 запросов при попытке

загрузить часть истории запросов составления меню. Во втором случае мы немало тратимся по памяти, но при этом получаем всю нужную информацию одним запросом. Мы посчитали, что в современных реалиях затраты объемов памяти хранения не так страшны(особенно с учетом чистки архива по истечению какого-либо срока, либо лимита хранимой истории), как постоянные запросы к БД и серверу соответственно. Если это будет делать большое количество человек одновременно, то могут возникнуть большие проблемы и сервер будет перегружен. В конце концов, нереляционные базы данных тем и хороши, что позволяют очень быстро и просто получать необходимые данные по заданным критериям.

Реляционная модель

С реляционной моделью данных все также было бы очень просто. Запрос выглядел бы следующим образом:

```
SELECT recipes.name, ..., recipes.steps_id FROM recipes WHERE recipes.calories >= caloriesLowerBound AND recipes.calories <= caloriesUpperBound AND recipes.cost >= caloriesLowerBound AND recipes.cost <= caloriesUpperBound AND recipes.type = type LIMIT 7;
```

Как мы видим, все тоже довольно просто. Какая-нибудь ORM, например Hibernate, преобразует полученную в результате запроса таблицу в java-объекты. Если бы у нас присутствовал архив в реляционной модели, то все было бы не многим сложнее, нежели в NoSQL. Запрос исполнялся бы либо через JOIN двух таблиц(архив, которые хранит список id рецептов).

==========

Итак, подведем некоторые итоги: как реляционная подход, так и нереляционный подход безусловно хороши. Иначе NoSQL сегодня не был так популярен, а SQL бы не использовался в огромном количестве коммерческих проектов. Но, как нам показалось, область применения NoSQL(под областью применения имеется ввиду ситуация, когда NoSQL подход позволит сделать работу с БД намного эффективнее) намного уже, чем у SQL. Те задачи, где будет иметь преимущество NoSQL, также могут быть решены с не особо большим отставанием и в реляционной модели, но наоборот в преимущественном большинстве случаев это не работает. NoSQL также очень хорошо справляется с теми задачами, когда данные не имеют высокой степени связанности, либо что еще лучше, данные в разных коллекциях вообще не связаны между собой. Также желательна некоторая статичность данных, где преимущественно происходит поиск информации, среди большого числа информации, в особенности, когда важна скорость доступа к этим данным.

P.S. Небольшая ремарка: наш проект - это далеко не показатель ситуации, когда выгодно использовать ту или иную конкретную модель, связано это с его относительной простотой. И разумеется, что все наши оценки весьма и весьма субъективны. Чтобы иметь более ясную картину, нужно иметь намного больше опыта работы с теми же SQL и NoSQL СУБД. К тому же всегда можно ознакомиться с мнением огромного числа разработчиков на различных сторонних ресурсах.

Литература

1. Sql. Полное руководство, Грофф Дж. Р., Вайнберг П.Н., Оппелъ Э. Дж. Вильямс, 2015, 960 с.

2. NoSQL. Новая методология разработки нереляционных баз данных, Мартин Фаулер, Прамодкумар Дж. Садаладж. Вильямс, 2013, 192

3. Официальная документация к NoSql DB: MongoDB. URL: https://docs.mongodb.com/v3.4/core/document/ (дата обращения: 01.12.17).

4. Официальная документация к NoSql DB: PostgreSQL DB. URL: https://www.postgresql.org/docs/ (дата обращения: 01.12.17)

А.А. Михайлов, студент, mi.mi.mikhaylov97@gmail.com
А.С. Васильйев, студент, alexandrvasilievby@gmail.com
Н.М. Селивёрстов, студент, nikitaseliverstov@gmail.com
(Санкт-Петербургский государственный электротехнический университет «ЛЭТИ», ул. Профессора Попова, 5, г. Санкт-Петербург, 197376, Россия)