

MongoDB (продолжение)

Отсутствие JOIN-ов

Мы должны делать JOIN-ы вручную, в коде своего приложения. По существу, мы должны делать второй запрос, чтобы найти связанные данные. Создание данных тут не сильно отличается от создания внешних ключей в реляционных базах.

Массивы и вложенные

документы

Но тот факт, что у MongoDB нет JOIN-ов еще не означает, что у неё не припасено пару козырей в рукаве. MongoDB поддерживает массивы, как объекты первого класса? Оказывается, что это очень удобно, когда требуется смоделировать отношения "один-ко-многим" или "многие-ко-многим". Например, если у сотрудника есть несколько менеджеров, мы просто можем сохранить их в виде массива:

```
db.employees.insert({_id: ObjectId("4d85c7039ab0fd70a117d733"), name: 'Siona',  
manager: [ObjectId("4d85c7039ab0fd70a117d730"),  
ObjectId("4d85c7039ab0fd70a117d732")] })
```

Денормализация

Еще одна альтернатива использованию JOIN-ов - денормализация. Исторически денормализация использовалась для оптимизации производительности, или когда с данных (например, журнала аудита) необходимо было иметь возможность делать снимок. Однако с быстрым ростом NoSQL решений, многие из которых лишены JOIN-ов, денормализация стала в порядке вещей. Это не означает, что нужно дублировать всё подряд в любых документах. Можно остерегаться дублирования данных, а можно соответствующим образом продумать архитектуру своей базы.

К примеру, мы разрабатываем форум. Традиционный путь ассоциировать пользователя с его постом - это колонка `userid` в таблице `posts`. с такой моделью нельзя отобразить список постов без дополнительного извлечения данных (JOIN) из таблицы пользователей. Возможное решение - хранить имя пользователя (`name`) вместе с `userid` для каждого поста. Можно также вставлять небольшой встроенный документ, например,

Мало или много коллекций

- Учитывая то, что коллекции не привязывают нас к конкретной схеме, вполне возможно обойтись одной коллекцией, имеющей документы разной структуры. Построенные на MongoDB системы, с которыми мне приходилось сталкиваться, как правило, были похожи на реляционные базы данных. Другими словами, то, что являлось бы таблицей в реляционной базе данных, скорее всего реализуется, как коллекция в MongoDB (таблицы-связки "многие-ко-многим" являются важным исключением).
- Дело принимает интересный оборот, если воспользоваться вложенными документами. Пример, который первым делом приходит на ум, это блог. Допустим, есть коллекция posts и коллекция comments, и каждый пост должен иметь вложенный массив комментариев. Если оставить в стороне ограничение 4Мб ("Гамлет" на английском едва дотягивает до 200 килобайт, насколько же должен быть популярным ваш блог?), большинство разработчиков предпочитают разделять сущности. Так понятнее и яснее.

MapReduce

MapReduce - это подход к обработке данных, который имеет два серьёзных преимущества по сравнению с традиционными решениями.

Первое и самое главное преимущество - это *производительность*. Теоретически MapReduce может быть распараллелен, что позволяет обрабатывать огромные массивы данных на множестве ядер/процессоров/машин. Как уже упоминалось, это пока не является преимуществом MongoDB.

Вторым преимуществом **MapReduce** является возможность описывать обработку данных нормальным кодом. По сравнению с тем, что можно сделать с помощью *SQL*, возможности кода внутри MapReduce намного богаче и позволяют расширить рамки возможного даже без использования специализированных решений.

Теория и практика

MapReduce - процесс двухступенчатый. Сначала делается *map* (*отображение*), затем - *reduce* (*свёртка*). На этапе отображения входные документы трансформируются (*map*) и порождают (*emit*) пары *ключ=>значение* (как *ключ*, так и *значение* могут быть составными).

При свёртке (*reduce*) на входе получается *ключ* и *массив значений*, порождённых для этого ключа, а на выходе получается финальный результат. Посмотрим на оба этапа и на их *выходные данные*.

Будем генерировать отчёт *по* дневному количеству хитов для какого-либо ресурса (например, веб-страницы). Это *hello world* для MapReduce. Для наших задач мы воспользуемся коллекцией *hits* с двумя полями: *resource* и *date*.

Желаемый результат - это отчёт в разрезе ресурса, года, месяца, дня и количества.

Пусть в `hits` лежат следующие данные:

<code>resource</code>	<code>date</code>
<code>index</code>	<code>Jan 20 2010 4:30</code>
<code>index</code>	<code>Jan 20 2010 5:30</code>
<code>about</code>	<code>Jan 20 2010 6:00</code>
<code>index</code>	<code>Jan 20 2010 7:00</code>
<code>about</code>	<code>Jan 21 2010 8:00</code>
<code>about</code>	<code>Jan 21 2010 8:30</code>
<code>index</code>	<code>Jan 21 2010 8:30</code>
<code>about</code>	<code>Jan 21 2010 9:00</code>
<code>index</code>	<code>Jan 21 2010 9:30</code>
<code>index</code>	<code>Jan 22 2010 5:00</code>

На выходе мы хотим следующий результат:

<code>resource</code>	<code>year</code>	<code>month</code>	<code>day</code>	<code>count</code>
<code>index</code>	<code>2010</code>	<code>1</code>	<code>20</code>	<code>3</code>
<code>about</code>	<code>2010</code>	<code>1</code>	<code>20</code>	<code>1</code>
<code>about</code>	<code>2010</code>	<code>1</code>	<code>21</code>	<code>3</code>
<code>index</code>	<code>2010</code>	<code>1</code>	<code>21</code>	<code>2</code>
<code>index</code>	<code>2010</code>	<code>1</code>	<code>22</code>	<code>1</code>

(Прелесть данного подхода заключается в хранении результатов; отчёты генерируются быстро и рост данных контролируется - для одного ресурса в день будет добавляться *максимум* один документ.)

Теперь сосредоточимся на понимании концепции.

- Первым делом рассмотрим функцию отображения. Задача функции отображения - породить значения, которые в дальнейшем будут использоваться при свёртке. Порождать значения можно ноль или более раз. В нашем случае - как чаще всего бывает - это всегда будет делаться один раз. Представьте, что map в цикле перебирает каждый документ в коллекции hits. Для каждого документа мы должны породить *ключ*, состоящий из ресурса, года, месяца и дня, и примитивное *значение* - единицу:

```
function() {  
  var key = {  
    resource: this.resource,  
    year: this.date.getFullYear(),  
    month: this.date.getMonth(),  
    day: this.date.getDate()  
  };  
  emit(key, {count: 1});  
}
```

this ссылается на текущий рассматриваемый документ. Надеюсь, результирующие данные прояснят для вас картину происходящего. При использовании наших тестовых данных, в результате получим:

```
{resource: 'index', year: 2010, month: 0, day: 20} => [{count: 1}, {count: 1}, {count:1}]
{resource: 'about', year: 2010, month: 0, day: 20} => [{count: 1}]
{resource: 'about', year: 2010, month: 0, day: 21} => [{count: 1}, {count: 1}, {count:1}]
{resource: 'index', year: 2010, month: 0, day: 21} => [{count: 1}, {count: 1}]
{resource: 'index', year: 2010, month: 0, day: 22} => [{count: 1}]
```

Понимание этого промежуточного этапа даёт *ключ* к пониманию MapReduce. Порождённые данные собираются в массивы *по* одинаковому ключу. *Java* разработчики могут рассматривать это как тип `HashMap<Object,ArrayList>` (Java).

Давайте изменим нашу *map*-функцию несколько надуманным способом:

```
function() {
  var key = {resource: this.resource, year: this.date.getFullYear(), month: this.date.getMonth(), day: this.date.getDate()};
  if (this.resource == 'index' && this.date.getHours() == 4) {
    emit(key, {count: 5});
  } else { emit(key, {count: 1}); }
}
```

Первый промежуточный результат теперь изменится

```
...
{resource: 'index', year: 2010, month: 0, day: 20} => [{count: 5}, {count: 1}, {count:1}]
```

Обратите внимание, как каждый *emit* порождает новое *значение*, которое группируется *по* ключу.

Reduce-функция

Reduce-функция берёт каждое из этих промежуточных значений и выдаёт конечный результат. Вот так будет выглядеть

```
function(key, values) {  
  var sum = 0;  
  values.forEach(function(value) {  
    sum += value['count'];  
  }); return {count: sum};  
};
```

На выходе получим:

```
{resource: 'index', year: 2010, month: 0, day: 20} => {count: 3}  
{resource: 'about', year: 2010, month: 0, day: 20} => {count: 1}  
{resource: 'about', year: 2010, month: 0, day: 21} => {count: 3}  
{resource: 'index', year: 2010, month: 0, day: 21} => {count: 2}  
{resource: 'index', year: 2010, month: 0, day: 22} => {count: 1}
```

Технически в MongoDB результат выглядит так:

```
_id: {resource: 'home', year: 2010, month: 0, day: 20}, value: {count: 3}
```

Это и есть наш конечный результат.

Почему мы просто не написали `sum = values.length`? Это было бы эффективным подходом, если бы мы суммировали массив единиц.

На деле `reduce` не всегда вызывается с полным и совершенным набором промежуточных данных. Например вместо того, чтобы быть вызванным с:

```
{resource: 'home', year: 2010, month: 0, day: 20} => [{count: 1}, {count: 1}, {count:1}]
```

Reduce может быть вызван с:

```
{resource: 'home', year: 2010, month: 0, day: 20} => [{count: 1}, {count: 1}]
```

```
{resource: 'home', year: 2010, month: 0, day: 20} => [{count: 2}, {count: 1}]
```

Чистая практика

Сперва давайте создадим набор данных:

```
db.hits.insert({resource: 'index', date: new Date(2010, 0, 20, 4, 30)});
db.hits.insert({resource: 'index', date: new Date(2010, 0, 20, 5, 30)});
db.hits.insert({resource: 'about', date: new Date(2010, 0, 20, 6, 0)});
db.hits.insert({resource: 'index', date: new Date(2010, 0, 20, 7, 0)});
db.hits.insert({resource: 'about', date: new Date(2010, 0, 21, 8, 0)});
db.hits.insert({resource: 'about', date: new Date(2010, 0, 21, 8, 30)});
db.hits.insert({resource: 'index', date: new Date(2010, 0, 21, 8, 30)});
db.hits.insert({resource: 'about', date: new Date(2010, 0, 21, 9, 0)});
db.hits.insert({resource: 'index', date: new Date(2010, 0, 21, 9, 30)});
db.hits.insert({resource: 'index', date: new Date(2010, 0, 22, 5, 0)});
```

Теперь можно создать *map* и *reduce* функции (консоль MongoDB позволяет вводить многострочные конструкции):

```
var map = function() {
  var key = {resource: this.resource, year: this.date.getFullYear(), month:
              this.date.getMonth(), day: this.date.getDate()};
  emit(key, {count: 1});
};

var reduce = function(key, values) {
  var sum = 0;
  values.forEach(function(value) {
    sum += value['count'];
  });
  return {count: sum};
};
```

Мы выполним команду `mapReduce` над коллекцией `hits` следующим образом:

```
db.hits.mapReduce(map, reduce, {out: 'hit_stats'});  
db.hit_stats.find();
```

Надпись `{out: 'hit_stats'}` означает, что результат сохраняется в коллекцию `hit_stats`:

Индексы

- В самом начале мы видели коллекцию `system.indexes`, которая содержит информацию о всех индексах в нашей базе данных. Индексы в MongoDB работают схожим образом с индексами в реляционных базах данных: они ускоряют выборку и сортировку данных. Индексы создаются с помощью `ensureIndex`:
- `db.unicorns.ensureIndex({name: 1});`
- И уничтожаются с помощью `dropIndex`:
- `db.unicorns.dropIndex({name: 1});`
- Уникальный индекс может быть создан, если во втором параметре установить `unique` в `true`:
- `db.unicorns.ensureIndex({name: 1}, {unique: true});`
- Можно создавать индексы над вложенными полями (опять же, используя точечную нотацию), либо над массивами. Также можно создавать составные индексы:
- `db.unicorns.ensureIndex({name: 1, vampires: -1});`
- Порядок вашего индекса (1 для восходящего и --1 для нисходящего) не играет роли в случае с простым индексом, однако он может быть существенен при сортировке или лимитировании с применением составных индексов.
- На [странице описания индексов](#) можно найти дополнительную информацию.

Шардинг

MongoDB поддерживает авто-шардинг. Шардинг - это подход к масштабируемости, когда отдельные части данных хранятся на разных серверах. Примитивный пример - хранить данные пользователей, чье имя начинается на буквы А-М на одном сервере, а остальных - на другом. Возможности шардинга MongoDB значительно превосходят данный простой пример

Репликация

- Репликация в MongoDB работает сходным образом с репликацией в реляционных базах данных. Записи посылаются на один сервер - ведущий (*master*), который потом синхронизирует своё состояние с другими серверами - ведомыми (*slave*). Вы можете разрешить или запретить чтение с ведомых серверов, в зависимости от того, допускается ли в вашей системе чтение несогласованных данных. Если ведущий сервер падает, один из ведомых может взять на себя роль ведущего.
- Хотя репликация увеличивает производительность чтения, делая его распределённым, основная её цель - увеличение надёжности. Типичным подходом является сочетание репликации и шардинга. Например, каждый шард может состоять из ведущего и ведомого серверов. (Технически, вам также понадобится арбитр, чтобы разрешить конфликт, когда два ведомых сервера пытаются объявить себя ведущими. Но арбитр потребляет очень мало ресурсов и может быть использован для нескольких шардов сразу.)

Статистика

Статистику базы данных можно получить с помощью вызова `db.stats()`. В основном информация касается размера вашей базы данных. Также можно получить статистику коллекции, например `unicorns`, с помощью вызова `db.unicorns.stats()`. Большая часть получаемой информации, опять же, касается размеров коллекции.