УДК 004.65

Последовательность как абстракция структурированного построения баз данных

Марчук А.Г. (Институт систем информатики СО РАН, Новосибирский государственный университет)

В статье представлена модель баз данных, основанная на последовательности объектов. Рассмотрены вопросы сериализации/десериализации объектов, индексных построений, структуры и методов универсальной последовательности и универсального индекса, реализация полного базиса редактирования через использование первичного ключа, временной отметки и «пустого» значения.

Ключевые слова: *база данных, последовательность, индексное построение.*

1. Введение

Классическим моделям, связанным с базами данных уже более 50 лет. В информационных технологиях закрепились реляционные базы данных, построенные на реляционной алгебре [1]. Однако, в последние два десятка лет активно развиваются и альтернативные, точнее дополняющие подходы, получившие обобщающее название NoSQL, которое чаще интерпретируют не как отрицание SQL, а как Not Only SQL [2]. Сейчас уже твердо вошли в практику, получили свою классификацию такие подходы, как документные базы данных, хранилища ключ-значение, семейства столбцов, графовые СУБД.

Предлагается модель структурного объекта, способного быть основой для разных построений баз данных. В принципе, это обобщение реляционных таблиц на случаи, когда не обязателен фиксированный набор колонок и скалярный тип этих колонок. Существенной частью модели является введение темпоральности, эквивалентности элементов, индексных построений. Это позволяет решать более широкий круг задач, чем в рамках табличного подхода. Кроме того, модель ориентируется на хранение/обработку больших данных, в частности на распределение базы данных по узлам, экономное решение задач избыточного хранения и восстановления данных, фиксации промежуточных состояний и откатов к предыдущим состояниям. Модель реализована в рамках системы PolarDB как библиотека классов в среде .NET и C#.

2. Что такое последовательность

Последовательность – это упорядоченный, конечный (ноль или более), растущий набор элементов:

```
e0, e1, e2, …, eN
ei – объекты
```

Частный случай последовательности, это Stream – растущая последовательность байтов обычно фигурирующая в процессах ввода/вывода. В общем случае элементы последовательности изменяемые, последовательность может иметь ноль элементов. В любой момент времени последовательность конечная. Последовательность может прирастать «вправо» через добавление элементов. Не предполагается, что можно добавлять элементы «слева» и исключать элементы из последовательности.

Элементы последовательности это структурные значения. Структурные значения – в данном случае это значения, выполненные в базисе построений той или иной системы программирования, включают в себя значения примитивных типов, таких как булевские, байты, символы, числа и составные значения. Мы ориентируемся на рекуррентное и иерархическое построение значений. В качестве примеров можно привести базис современной объектно-ориентированной системы программирования, напр. .NET, Java. Более специализированными построениями напр. являются структурный базис JSON (значения, записи, массивы) и DOM – внутреннее представление объектов XML. Также важным использованным классом является структуризация системы Polar [3].

Нас будут интересовать «большие» последовательности. Большие последовательности (или в общем случае, большие данные) в нашем случае, это такие информационные образования, которые невозможно или нерационально располагать в оперативной памяти. А это, в частности означает, что должна существовать форма представления последовательностей для оперативной работы с ними. Таких форм предлагается две: в виде потоков объектов (генерация потока, фильтрация, функциональное преобразование, группирование, редукция, использование) в функциональном слое программирования, напр. LINQ в C#, Scala или Flink в Java и в виде отображения последовательности объектов на устройства внешней памяти. В примерах потоковая обработка будет иллюстрирована средствами C# и LINQ [4]. А отображение на внешнюю память реализуется через сериализацию/десериализацию.

3. Сериализация и десериализация последовательностей

Сериализация кодирует структурное значение последовательностью алфавита кодирования, десериализация делает обратное преобразование последовательности символов кодирования в объектное представление. Алфавит кодирования может быть битами, байтами, символами.

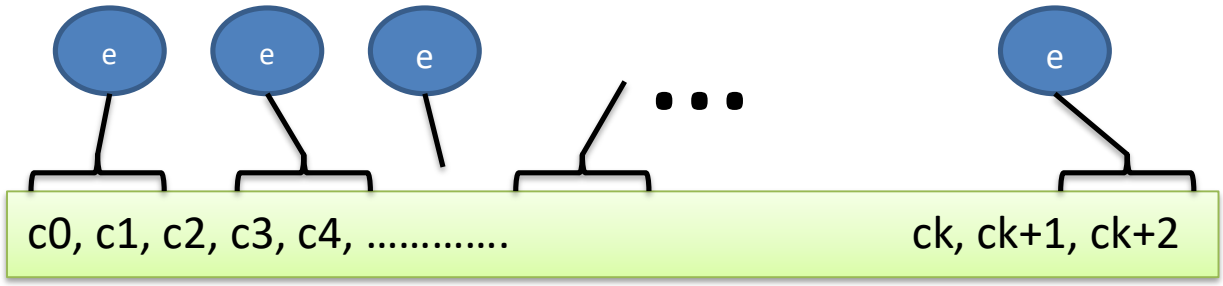

Рис. 1. Отображение последовательности элементов на цепочку символов и обратно

На рисунке 1 иллюстрируется отображение последовательности объектов (овалы сверху) на цепочку символов какого-то алфавита. Элементы основной последовательности преобразуются в цепочки символов этого алфавита, цепочки укладываются в результирующую последовательность подряд, возможно, через разделители. Число символов в цепочке, соответствующих одному элементу основной последовательности – произвольное. Обратный процесс является десериализацией.

Хорошая сериализация допускает десериализацию с получением эквивалентной последовательности. Эквивалентность последовательностей и значений определяется рекуррентно.

В общем случае, десериализация требует синтаксического анализа. Также заметим, что последовательность может рассматриваться как единый объект – массив объектов, обычно сериализация последовательности выполняется по тем же правилам, что и сериализация элементов.

3.1 Текстовая сериализация

Текстовая сериализация отображает объект или последовательность объектов. Алфавитом служат символы. Наиболее часто используемая кодировка символов – юникод. Обычно, речь идет о кодировании структурных значений, иногда довольно специфичного вида. Для кодирования структурных данных используется инфиксная рекуррентная форма вида: открывающая скобка – значения, перечисленные через разделитель, закрывающая скобка. К этому виду текстового изображения структурных значений относятся языки разметка, в

первую очередь – XML, JSON и некоторые другие. Есть специализированные построения подобных текстовых структур, напр. LISP.

```
<db>
  …
  <person>
    <firstName>Иван</firstName>
    <lastName>Иванов</lastName>
    <address>
      <streetAddress>
        Московское ш., 101, кв.101
      </streetAddress>
      <city>Ленинград</city>
      <postalCode> 101101
      </postalCode>
    </address>
    <phoneNumbers>
      <n> 812 123-1234 </n>
      <n> 916 123-4567 </n>
    </phoneNumbers>
  </person>
  …
</db>
```

```
[ …
  {   "firstName": "Иван",
      "lastName": "Иванов",
      "address":
      {
          "streetAddress": "Московское
ш., 101, кв.101",
          "city": "Ленинград",
          "postalCode": 101101
       },
      "phoneNumbers": [ "812 123-
1234", "916 123-4567" ]
  }  …
]
```

Таблица 1. Образцы текстовых сериализаций в формате XML и JSON

Текстовая развертка (сериализация) структурных значений используется для случаев, когда нужно не только сохранить значение объекта или объектов, но и иметь его доступным для изучения и редактирования. Для целей среды сохранения данных в базах данных, такой способ неудобен по двум причинам: неэкономное использование байтового пространства и, главное, потребность в синтаксическом анализе при вводе (десериализации).

В ячейках таблицы 1 приведены примеры текстов XML и JSON, показывающих как выглядят записи условной базы данных. Видно, что JSON выигрывает и в компактности и в наглядности.

3.2 Байтовая сериализация

Для баз данных, наиболее естественной и эффективной, является байтовая сериализация. Обычно используется префиксная (польская) запись структуры: указание вида структуры или структуризатора, количество аргументов (если надо), подряд стоящие изображения аргументов. Такой подход годится как для общего безтипового случая, так и для типизированной записи значений. Основа бинарного байтового отображения структур зафиксирована в наборах действий с байтовыми потоками (Stream) в любой развитой системе программирования через класс BinaryWriter и класса чтения BinaryReader. Напр.

```
BinaryWriter bw = new BinaryWriter(file);
bw.Write(false);
```

```
bw.Write((byte)121);
bw.Write(999);
bw.Write(“Sample string”);
```

Чтение с того же места той же последовательности типов значений позволяет получить значения, эквивалентные тем, которые записывались, Введение в эту схему рекуррентного применения записи в случае массива объектов требует лишь дополнительной записи количества элементов:

```
bw.Write(nelements); bw.Write(el1); bw.Write(el2); …
```

Проблема в том, что если мы не знаем типа читаемого элемента, мы можем прочитать «не то». Соответственно, для безтиповой последовательности, запись каждого значения должна предваряться записью варианта его типа.

```
Object x =…
x is bool      t b
x is byte      t b
x is int       t bbbb
x is string    t bssssss
x is Object[]  t bbbbbbbb (e0) (e1) …
```

t – номер варианта (0-bool, 1-byte, 2-int, 3-string, 4-array

b – байт значения

s – байт значения строки

e – развертка элемента массива (скобки для подчеркивания рекуррентного раскрытия элементов.

Сериализация для типизованных значений строится по этой же схеме. Только отсутствует колонка t – номеров вариантов. Это потому, что имеющаяся внешняя информация о типах определяет номер варианта. Соответственно, при раскрытии элементов, их тип вычисляется и снова не требует фиксации номера.

3.3 Битовая сериализация

При такой сериализации осуществлется отображение объектов на цепочки битов. В принципе, идейная часть сериализации аналогична байтовой. Существенная разница в том, что координаты позиции измеряются в числе битов от начала последовательности и, кроме того, надо активно использовать теорию кодирования для достижения оптимальных решений. Рассмотрим пример (в синтаксисе языка Поляр):

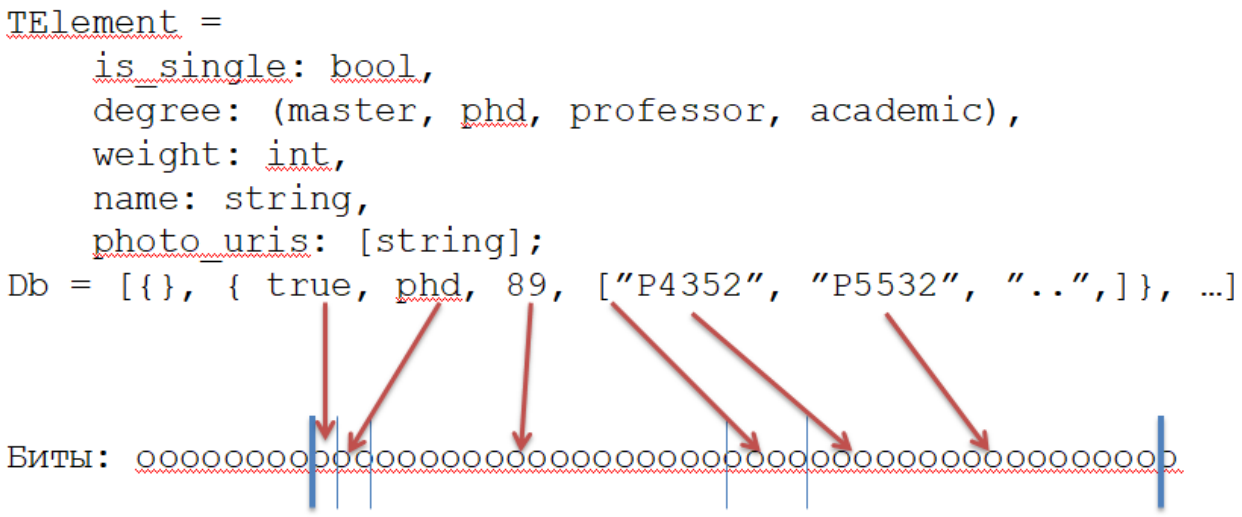

Рис. 2. Пример отображения полей записи на битовую последовательность

На рисунке 2 отдельные биты обозначены кружочками и условно демонстрируются следующие эффекты битовой сериализации: а) булевское значение можно отображать в 1 бит; 2) перечислимое значение второго поля отображаются в минимальное количество бит, зависящее от числа вариантов; 3) целочисленное значение динамически кодируется так, чтобы минимизировать применяемое число бит; 4) то же самое касается указания количества элементов в подпоследовательности.

3.4 Позиции элементов в последовательности

Как уже отмечалось, (большая) последовательность существует либо в виде динамического потока элементов, либо в виде сериализации. А фактически, после первичной загрузки данных – в виде сериализации. Позицией элемента в сериализации назовем число байтов (битов, в случае битовой сериализации) от начала последовательности до начала элемента. Пара сериализация-десериализация должна строиться так, чтобы элементы корректно прочитывались если правильно указана позиция элементов и тип, в случае типизированной последовательности. Для текстовой сериализации можно было бы вести подсчет символов от начала, но практичнее – количество байтов, поскольку позицию можно использовать для «прямого» доступа к элементу последовательности (через метод seek или свойство .Position, определенных для класса Stream). Для битовой развертки последовательности, использование позиции несколько усложняется, хотя принципиально такое же.

В дальнейшем, мы будем говорить об элементе последовательности как о паре: (элемент, позиция). Более того, элемент последовательности теперь определен через позицию. По

позиции из сериализации прочитывается элемент и снова получается пара. Введем несколько обозначений. Сериализации будем обозначать **S**, элементы последовательности **e**, позицию конкретного элемента **e** как e_{pos}**.** Просто позицию обозначим **p**, через позицию вычисляется сам элемент **e = S(p)**.

Позиции можно использовать в различных информационных построениях. Например, если сформировать массив позиций **arr**, то **i**-ый элемент последовательности будет **S(arr[i])**.

Стоит отметить, что позиция (в нашем случае) реализуется длинным целым. То есть, значением с фиксированной длиной в сериализации. Соответственно, для реализации прямого доступа к элементам последовательности (по индексу) можно пользоваться не только массивом, но и последовательностью позиций, обозначим последовательности позиций через **P**. Тогда по индексу **ind** можно найти позицию в **P** как

```
offset = число_байтов_в_начале + 8 * ind
(8 – число байтов в целом двойной точности)
p = P(offset)
e = S(p)
```

4. Индексное построение

Рассмотрим задачу поиска элемента в последовательности. Точнее, элементов, удовлетворяющих какому-то критерию. Пусть **X** – множество элементов последовательности, поиск по критерию:

```
{ x ∈ X | C(x) }, параметрический вариант { x ∈ X | Q(x, y) }
```

В общем случае, задача решается перебором элементов с проверкой критерия на каждом. В частных случаях, можно выстроить дополнительную информационную структуру, которая сделает поиск более экономным. Эту структуру будем называть индексным построением (database index).

4.1 Ключевой индекс

Пусть на элементах **X** определен функционал **F(x)**, а критерием будет **Q(x, y) : F(x) = y**. Тогда достаточно для каждого элемента из **X**, создать пару **(F(x),** x_{pos}**)**. Множество всех таких пар отсортируем по первому значению. Тогда поиск можно выполнять как бинарный поиск во множестве пар, по найденным парам вычисляем элементы-результаты поиска. На рисунке 3 приведена иллюстрация формирования и использования ключевого индекса для поиска по заданному значению.

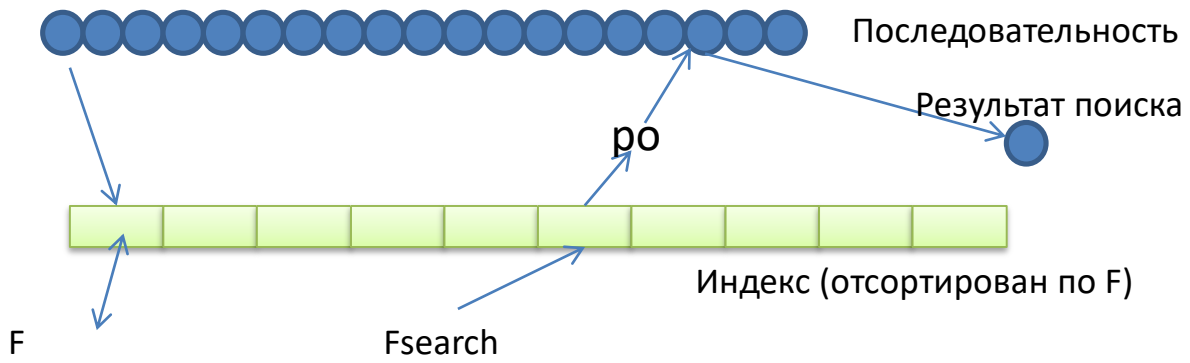

Рис. 3. Ключевой индекс и его использование для поиска по значению

Из последовательности выделяются позиции элементов, формируется последовательность пар, далее, эта последовательность сортируется по первому значению пары, индекс построен. Использование индекса выполняется через бинарный поиск всех пар, в которых первое значение совпадает с образцом F_{search}. Далее, из выявленных элементов берутся позиции и по ним, получаются результаты поиска. Функционал `F(x)` называется в этом случае ключевой функцией.

Такая схема не получается в случае, если тип значений ключевой функции не дает фиксированного размера в сериализации. Например, часто встречающаяся ключевая функция – строковый идентификатор именно такой. Схема изменяется на более простую. В индекс помещаются только позиции элементов. При этом сортировка ведется на элементах, вычисленных по позиции. Соответственно изменяется процедура поиска, в которой бинарный поиск выполняется также на вычисленных через позицию элементах. Такая косвенность заметно «утяжеляет» и построение индекса и поиск по индексу.

4.2 Использование компаратора

Главное в сортировочном индексе, это упорядочение элементов в соответствии с отношением порядка, заданного на элементах. Для объектов, достаточно общей случай отношения порядка задается компаратором. Компаратор, это функция, определенная на множестве значений элементов последовательности такая, что

```
int Compare(object a, object b)
```

Значения компаратора: -1, 0, 1, соответствующие разным вариантам отношения, кроме того, должны удовлетворяться метрические аксиомы, напр. транзитивность.

Использование компаратора выполняется следующим образом. Для подготовки индекса по сериализованной последовательности строится массив позиций, массив сортируется с

использованием компаратора. При использовании формируется элемент-образец `sample`, далее бинарным поиском находятся все элементы последовательности, для которых

```
Compare(sample, x) = 0
```

эти элементы составляют решение.

Имеется некоторое неудобство проведения поиска через компаратор: это необходимость критерий поиска превращать в целостный элемент. Как правило, это не вызывает проблем и эффективность поиска по-прежнему остается высокой.

4.3 Ускорители для индексов

Что из себя представляют индексы? Логически, индексы, это последовательности позиций, или пар (позиция, значение). Поскольку последовательности предполагаются большими, то и индексы будут большими. А значит, их следует разворачивать также в сериализации таких значений. Позиции (длинные целые) – значения одинакового размера (8 байтов), если добавляется числовое значение, то такая запись также фиксированного размера. Это позволяет вычислять позицию элемента во вспомогательной индексной сериализации по его номеру и полноценно использовать элементы для поисков по значению или по компаратору.

Иногда желательно усложнить индексное построение для того, чтобы ускорить поиск элементов. На уровне объектной абстракции, предлагается два варианта построения ускорительных дополнительных построений.

Прореженный массив

Предположим, есть индексный массив (сериализация) уже упорядоченный в соответствии с критерием данного индекса, т.е. либо в соответствии с компаратором, либо в соответствии с частным случаем компаратора – ключевой функцией. Тогда сделаем выборку значений из массива с равномерным шагом, как это изображено на рисунке 4.

Рис. 4. Разреженная выборка значений из отсортированного массива позиций

Выбираются значения, участвующие в упорядочивании по заданному критерию. Если в массиве находятся только позиции, как в случае компараторного индекса, то по смещениям прочитываются элементы основной последовательности. При наличии построенного

дополнительного массива, решение основной задачи – поиску по образцу, производится сначала в дополнительном массиве, который отсортирован оп построению, а потом в поддиапазоне основного индекса.

Существует интересный вариант формирования прореженного массива. Пусть число элементов индексного построения являются степенью двойки и пусть отсортированные элементы индексного построения p_i переупорядочиваются в массив q_j таким образом, что новый индекс **j** является битовой инверсией исходного индекса **i**. То есть, если **i** состоит из битов **b0, b1, b2, … bk**, то инверсный индекс будет **bk, bk-1, …, b0**. В этом построении, элемент q_0 будет соответствовать точке **N/2**, т.е. середине массива. Следующие два элемента будут соответствовать серединам первой половины и второй половины и т.д. То есть, эти начальные, также как и следующие точки, будут повторять логику двоичного поиска. Если **K**, степень двойки, элементов инверсного массива построить отдельно, это будет прореженный массив с фиксированным интервалом между элементами.

Создание шкалы значений

Другой способ реализуется для числовых ключей, значения которых более или менее распределяются в диапазоне **[min, max]**. Этот диапазон разбивается на интервалы равномерным шагом по значению и фиксируются номера элементов массива индексных значений, соответствующие началам интервалов.

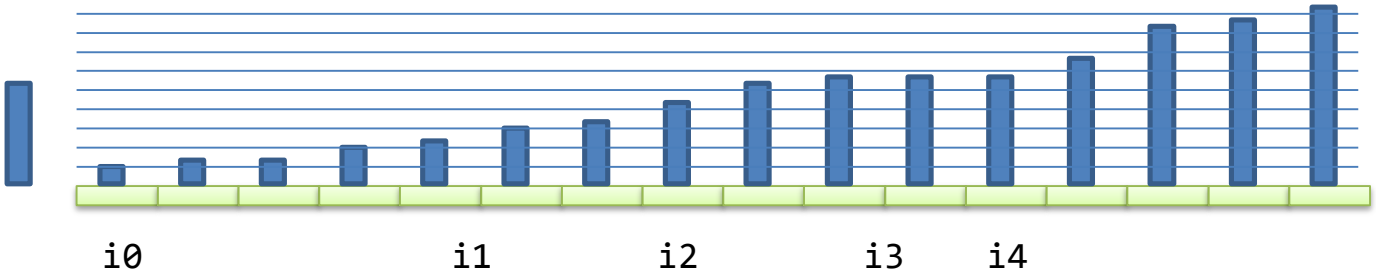

Рис. 5. Шкалирование упорядоченных числовых значений

Теперь, когда появляется ключевое значение (левый столбик), поисковая процедура определяет какому интервалу это значение принадлежит и границы интервала становятся интервалом поиска группы значений, имеющих тот же ключ.

4.4 Эффективность использования индексного построения

Рассмотрим производительность нашего построения последовательности с заданным критерием и построенным индексом. Пусть есть случайный набор ключевых значений или поисковых образов. Нас будет интересовать средняя скорость выполнения выборки по критерию.

Сначала обратим внимание собственно на сериализацию последовательности. Предположим, мы таинственным образом знаем позиции элементов, которые надо найти. Остается только прочитать (десериализовать) значение. Эту ситуацию можно промоделировать. Для этого запишем в файл с помощью BinaryWriter массив длинных целых или чего-то аналогичного. А потом будем делать выборку этих значений по случайному номеру по позициям, которые легко вычисляются.

Массив заполняется довольно быстро: 100 млн. значений загружаются около 2.5 сек. Прочтение всех записанных величин через бинарное чтение «подряд» выполняется приблизительно за то же самое время. А прочтение элементов по случайно заданному номеру, выполняется 7 миллисекунд на 1000 чтений. И эта скорость сохранится если мы отключим заполнение файла поскольку он уже сформирован. Однако, если теперь мы перезагрузим компьютер и снова запустим тест скорости случайных выборок, то результат может обескуражить: будет около 100 выборок в секунду. Это из-за того, что современные операционные системы кешируют в оперативной памяти не только страницы программы, но и страницы файлов. А это означает, что если кеш не работает (не успел накопиться), то доступ к диску будет выполняться в соответствии с характеристиками устройства (7200 rsp для «стандартного»), т.е. порядка 100 выборок в секунду. Использование SSD несколько ускорит этот процесс, но качественно не изменит.

Таким образом, без кеша скорость доступ порядка 100 выборок в сек., с кешем – порядка 100 тыс. выборок в сек., для массива – это порядка 100 млн. выборок в сек.

Для больших данных ситуация с замедлением усугубляется. Если данные не помещаются в оперативную память, а поток запросов равномерен по пространству номеров, то не существует разумной политики страничного кеширования, которая обеспечит высокую (100 тыс.) скорость работы.

В любом случае, в алгоритмах выборки значений по критерию важнейшим фактором является количество чтений из больших массивов, отнесенное на одну выборку. Обычная схема алгоритма следующая: бинарный поиск в индексном массиве и выдача найденных значений. Если в индексном массиве записано ключевое значение элемента, то общее количество выборок **`lg2(N) + 1`**. Если в индексном массиве позиции, тогда для проверки значения выполняется одна выборка из индексного массива, это позиция, и одна выборка из основной последовательности. Общее количество выборок **`2xlg2(N)`**, где **`N`** – число элементов в последовательности.

Рассмотренные ускорители создают дополнительные к базовому отсортированному массиву структуры, но в отличие от основного индекса, ускоритель может иметь разный

размер. Ускорители могут быть реализованы в виде сериализации последовательности или в виде массива в оперативной памяти. Поиск производится сначала в ускорителе. Для прореженного массива поиск выполняется за **`lg2(K)`** шагов, где **`K`** – его размер. Для шкалы поиск в шкале выполняется за одну операцию. Таким образом, выигрыш от ускорителя, реализованного в виде сериализованной последовательности, будет только в варианте шкалы.

Если ускоритель реализуется в виде массива в оперативной памяти, выигрыш будет при любом ускорителе. Эффект он него будет зависеть от выделенного ресурса ОЗУ. Поскольку работа с объектами в оперативной памяти производится существенно быстрее работы с сериализованными объектами, время поиска можно оценить как **`lg2(N/K)`** или **`lg2(N) - lg2(K)`** выборок. Например, для большой последовательности напр. в 1 млрд. элементов число поисковых выборок из индексного массива может быть более 30. Наличие в ОЗУ массива в 1 млн. ускорительных значений может уменьшить это количество до порядка 10. При этом, использование ресурса ОЗУ может оказаться разумным.

Схема с ускорителем, размещаемым в ОЗУ, имеет недостатком не только использование дефицитного ресурса ОЗУ, но и то, что требуется время для загрузки вспомогательного массива. Загрузка существенно ускорится, если прореженный массив будет формироваться из инверсного массива (см. ранее). Точки прореженного массива будут находится рядом и подряд. Для чтения массива из одной (серединной) точки, нужно прочитать первое значение. Прочитав два следующих, прореженный массив будет из трех точек, следующие 4 значения снова его увеличивают и т.д. При этом, расположение точек «подряд» существенно ускоряет их чтение из файла.

5. Объектная модель последовательности

Создадим универсальную последовательность объектов сначала на тех построениях, которые были сделаны ранее по тексту статьи. В данном разделе мы будем придерживаться моделей и синтаксисом языка программирования C#, хотя все подходит и для других средств.

5.1 Структуризация данных

Используемые в построениях структуры данных традиционны для многих языков программирования и реализованы в большинстве систем программирования. Объектом являются булевское, байт, символ, различные числа, строки символов. Также объектом

является массив объектов. Последнее порождает рекуррентное построение, позволяющее моделировать широкий спектр иерархических структуризаций.

Типом данных назовем структурное построение, задающее кодирование/декодирование структурного значения в некотором базисе. В нашем случае, тип будет определять бинарную байтовую сериализацию/десериализацию структурных значений.

Здесь мы будем придерживаться типовой системы языка программирования Поляр [3, 5]. Типизованное значение представляет собой

```
- none (тип, с пустым множеством значений)
- bool
- byte
- char
- int
- long
- double
- запись
- массив
- объединение
```

Последние три – композиционные типы, **`запись`** – это набор полей заданных типов, **`массив`** – это набор элементов заданного типа, **`объединение`** – значение одного из заданных вариантов типов.

При сериализации тип none переходит в 0 байтов, типы `bool` и `byte` – 1 байт, `char` – 2 байта, `int` – 4 байта, `long` – 8 байтов, `double` – 8 байтов. Запись переходит в подряд стоящие сериализации полей, массив переходит в длинное целое, определяющее длину последовательности `N`, а потом `N` значений определенного типа. Объединение представляется номером текущего варианта (1 байт) из заданных вариантов типа, далее идет значение этого варианта.

Ранее мы допускали отсутствие заданного типа, которое при сериализации интерпретировалось как некоторый код-номер, а далее сериализация значения. Такое отсутствие типа можно трактовать как «автоматический» или «динамический» тип. Можно определить и этот («безтиповый») тип в нотации Поляра:

```
Dyna = isnull^none,
    Bool, Byte, Char, Int, Long, Double^none,
    Arr^Dyna;
```

Конкретные типовые определения будем предполагать экземплярами класса PType. Заметим, что общая объектная конструкция не сильно зависит от применяемой системы типов, поскольку конструкция объекта зафиксирована, а отображение значения на байты важно лишь обратимостью операции сериализация – десериализация.

5.2 Хранилище потоков

Еще одной составной частью объектной модели последовательности является хранилище потоков. Его задача – предоставлять байтовые потоки (Stream) для использования в модели. Потоки байтов требуются во всех построениях, где фигурирует сериализация.

Хранилище потоков может быть простым типа директории файловой системы с множеством файлов-потоков (FileStream). Но может быть и довольно сложным, в котором оптимизируется скорость доступа к потокам и кеширование, создается система контрольных сумм и резервного копирования.

Например, в библиотеке PolarDB [5] хранилище байтовых потоков формируется на основе специально разработанного классы PagedStream, расширяющего класс Stream, реализующего «свою» страничную память и псевдофайловую систему, похожую на первые варианты ОС Unix. Все потоки укладываются в страницы, на который разделен опорный файл, страницы имеют свою систему кеширования, возможно использование резервного копирования.

5.3 Универсальная последовательность

Сформируем универсальную последовательность объектов по схеме, изложенной ранее. Типизованная или безтиповая последовательность объектов сериализуется в бинарный байтовый поток (стрим). В стриме сохраняются: количество элементов в последовательности и все элементы в виде сериализаций. В силу обратимости сериализации через десериализацию, полученный стрим можно снова развернуть в поток объектов. Также для универсальной последовательности определен метод добавления единичного объекта. Этот метод влияет не только на «концовку» стрима, но и на начало, где корректируется текущая длина. Общая схема класса следующая:

```
public class UniversalSequence
{
    public UniversalSequence(PType tp_elem, Storage store) { … }
    public void Load(IEnumerable<object> flow) { … }
    public void AppendElement(object element) { … }
    public IEnumerable<object> Elements() { … }
}
```

Такая простая конструкция (имеется некоторое упрощение объектной модели) позволяет организовать довольно сложные построения с индексами и операциями редактирования последовательности. Это будет показано дальше.

5.4 Универсальный индекс

Охватить возможное множество индексных построений вряд ил возможно. В данной модели предлагается один гибкий вариант. В принципе, что нужно? Нужно выстроить массив позиций в соответствии с заданной упорядоченностью. В дальнейшем, этот массив используется для бинарного поиска по этой функции. Соответственно, класс индекса должен содержать ссылку на хранилище, чтобы сериализовать свои построения, ссылку на последовательность, назовем ее опорной, для которой этот индекс будет создаваться, функцию компаратора, задавать упорядочивание. Кроме того, индекс должен быть построен, но также должен и изменяться с изменением (ростом) опорной последовательности. Еще одни нюанс: индекс не обязан быть задан на всех элементах опорной последовательности, это можно регулировать функцией применимости. Удовлетворив требованиям, мы получим что-то вроде:

```
public class UniversalIndex
{
    public UniversalIndex(PType tp_elem, Storage store,
      UniversalSequence tab, Func<object, bool> applicable,
      Comparer<object> comp, Func<object, int> keyFun)
    { … }
    internal void OnBuild() { … }
    internal void OnAppendElement(object element, long pos) { … }
    public IEnumerable<object> Get(object sample) { … }
}
```

Некоторая дополнительная особенность класса в том, что кроме компаратора, мы объявляем ключевую функцию keyFun. Логика здесь такая, что упорядочивание осуществляется сначала по ключевой функции, а при одинаковости значений функции по компаратору. Часто ключевая функция играет роль хеш-функции. Допустимо, чтобы ключевая функция может быть не определена (**=null**) и наоборот, компаратор также может быть не определен.

Индекс работает в связке с опорной последовательностью. Опорная последовательность заполняется, потом строится (добавляем метод Build()), при построении строятся также индексы через запуски индексных методов OnBuild(). При построении индекса происходит формирование массива позиций и, возможно, ключевых значений, массив сортируется по указанным в индексе критериям и обслуживает запросы Get. В случае выполнения в последовательности метода AppendElement(e), запускается обработчик OnAppendElement в каждом из индексов.

5.5 Растущая модель

Сформированная в предыдущем разделе модель индекса, при прямолинейной реализации, не позволяет эффективно выполнять добавление элемента, поскольку при каждом добавлении нужно снова строить, а значит – сортировать индексный массив. Также пока нет механизма редактирования, т.е. не только добавления, но и уничтожения и изменения элементов в последовательности. Требуется более сложная модель индекса!

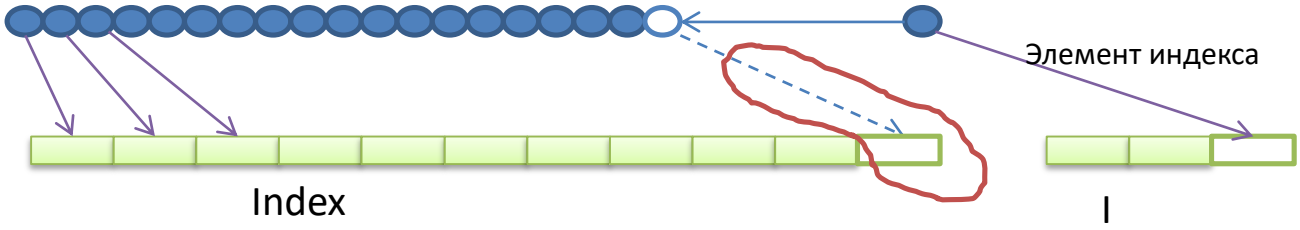

Рис. 6. Разбиение индексного массива на два

На рисунке 6 имеется иллюстрация усложненного устройства индекса на примере выполнения операции добавления элемента. Элемент (синий кружок справа) добавляется к последовательности через штатный метод AppendElement(e). Почему старая схема строения индексного построения не подходит? Потому что добавление в индексный массив (пунктирная стрелка) должно приводить к пересортировке всего индексного массива, что слишком затратно. Предлагается внести изменения в индексное построение тем, что индексное построение сделать из двух массивов: левого, большого, который при отработке индексного метода OnAppendElement не меняется и правого, маленького, в который идет добавление позиции и, возможно ключевого значения, с пересортировкой, что для малых количеств добавленных элементов не представляет проблемы.

Теперь рассмотрим задачи добавления объекта и редактирования объекта. В этом случае для начала, нужно определить способ указания какой именно элемент подвергается изменению.

Один из индексов можно определить как первичный ключ (primary key). Это выделение дает возможность произвести факторизацию по первичному ключу. То есть, элементы, совпадающие по первичному ключу объявляются эквивалентными. Среди эквивалентных, по некоторому критерию, назначается оригинал. Используемый критерий: оригиналом является элемент у которого было более позднее попадание в последовательность! Реализация критерия выполняется или через позиции эквивалентных элементов в последовательности или через добавление временной отметки.

Свойства первичного ключа: первичный ключ дает уникальное значение (по построению);

первичный ключ, как правило, реализуется через ключевое значение (число или строка).

Теперь для выборки можно добавить (и эффективно реализовать) метод единичной выборки по ключу

```
object element = GetByKey(object key);
```

Вернемся к задаче редактирования базы данных. Изменение элемента выполняется добавлением другого значения элемента с тем же первичным ключом. Уничтожение элемента выполняется через изменение на «пустое» значение.

6. Заключение

В статье представлена модель последовательности объектов, разработанная для использования в универсальных и специализированных построениях баз данных и систем управления базами данных.

Модель опирается на: сериализацию объектов; выделение позиции элемента как способа прямой работы с элементами; построение специальной конструкции индексов для реализации эффективной схемы отслеживания добавления элементов; назначение первичного ключа среди определенных индексов; введение на этой основе эквивалентности элементов и средств выявления оригинала, напр. через временную отметку; определение пустого элемента с заданным первичным ключом; реализацию полного набора операций редактирования через добавление элементов.

Модель обобщает классические реляционные таблицы, key-value хранилища, а также ряд NoSQL подходов таких как MondoDB, Cassandra, BigTable и др.

Модель была реализована в библиотеке PolarDB [5]. Эта реализация была использована в полупромышленных проектах фактографических систем ИСИ [6].

Список литературы

1. Дейт К. Дж. Введение в системы баз данных = Introduction to Database Systems. — 8-е изд. — М.: Вильямс, 2005. — 1328 с. — ISBN 5-8459-0788-8 (рус.) 0-321-19784-4 (англ.).
2. Мартин Фаулер, Прамодкумар Дж. Садаладж. NoSQL: новая методология разработки нереляционных баз данных = NoSQL Distilled. — М.: «Вильямс», 2013. — 192 с. — ISBN 978-5-8459-1829-1.
3. Марчук А.Г., Лельчук Т.И. Язык программирования Поляр: описание, использование, реализация, Новосибирск, 1986, 96 с.

4. Синтаксис LINQ = https://docs.microsoft.com/ru-ru/dotnet/csharp/linq/
5. Марчук А.Г Архитектура и основные особенности библиотеки PolarDB работы со структурированными данными // Системная информатика, № 13, 2018. Стр. 25-34
6. А.Г.Марчук, С.В.Лештаев Электронный архив газет: Web-публикация, ассоциация информации с базой данных, создание полнотекстового поиска // Аналитика и управление данными в областях с интенсивным использованием данных, XVIII Международная конференция DAMDID/RCDL'2016, Ершово, Московская обл., Россия, 11-14 октября 2016 года, Труды конференции. Торус пресс, Москва, 2016. Сс. 155-160.
7. А.Г.Марчук, П.А.Марчук Платформа реализации электронных архивов данных и документов // Электронные библиотеки: перспективные методы и технологии, электронные коллекции: Труды XIV Всероссийской научной конференции RCDL'2012. Переславль-Залесский, Россия, 15-18 октября 2012 г. – г. Переславль-Залесский: изд-во «Университет города Переславля», 2012, С. 332-338.