УДК 004.6

Архитектура и основные особенности библиотеки PolarDB работы со структурированными данными

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

В статье представлен анализ созданной в ИСИ СО РАН библиотеки PolarDB работы со структурированными данными. Это – библиотека классов, реализованная на платформе .NET Core. Она предназначена для создания систем структурирования, хранения и обработки данных, в том числе большого объема. Библиотека построена на ранее разработанной системе рекурсивной типизации и охватывает ряд существенных задач, таких как структурирование данных, сериализация, отображение данных на байтовые потоки, индексные построения, блочная реализация динамических байтовых потоков, распределение данных и обработки данных, резервное копирование и восстановление. Применение библиотеки PolarDB позволяет создавать эффективные решения для специализированных баз данных в разных парадигмах: последовательности, реляционные таблицы, key-value хранилища, графовые структуры.

Ключевые слова: *структурированные данные, типизация данных, базы данных, PolarDB, Big Data.*

1. Введение

При построении информационных систем, практически единственным решением относительно базы данных, является применение той или иной универсальной СУБД. В Институте систем информатики СО РАН был предложен и реализован подход, связанный с программированием требуемых решений средствами библиотеки. Подход был сформирован на основе системы типизации, имеющейся в языке программирования Поляр [1]. Начальная часть библиотеки формировалась в рамках проектов по исторической фактографии [2]. Первый рабочий вариант собственно системы PolarDB был представлен в докладе [3]. Подход и библиотека были опробованы при реализации некоторых системных и прикладных проектов [4, 5]. Библиотека используется в учебном курсе «Методы и технологии обработки больших данных», созданном по заказу АО «ВымпелКом» при поддержке Министерства науки и образования Новосибирской области.

2. Концептуальный базис

Структурное значение (structured value, поляровское структурное значение, p-value) - древовидное построение (значение), интерпретируемое в соответствии с заданным типом.

Тип (Type, поляровский тип, p-type) – древовидное построение, задающее интерпретацию для структурных значений. Может быть выражено как структурное значение. Тип может быть примитивным (атомарным) или конструируемым (составным).

Объект базы данных – сформированное в некотором рабочем поле или в хранилище структурное значение.

Объектное представление структурного значения – внутренняя конструкция в ОЗУ, вместе с типовым значением задающая структурное значение. Объектное представление реализуется средствами какой-то (напр. .NET) системы программирования, доступно через язык программирования и служит связующим средством между программными объектами и объектами базы данных по следующей схеме: объект базы данных полностью или частично, отдельными полями, может быть реализован значениями объектного представления. Соответственно, возможно "чтение" объектов базы данных или их полей в объектное представление. И наоборот, возможна "запись" значений объектного представления в объекты базы данных или их поля.

Текстовая сериализация (текстовое представление) структурного значения – точные правила представления любого структурного значения последовательностью символов. Текстовое представление, совместно с явно или неявно определенным типом этого представления, изображает структурное значение

Бинарная (байтовая) сериализация структурного значения – правила представления любого структурного значения в виде потока байтов. Бинарная сериализация осуществляется только в контексте типа сериализуемого или десериализуемого структурного значения.

Примитивные типы:

boolean, character, integer, longinteger, real – логическое, символьное, целое (32 разряда), длинное целое (64 разряда), число с плавающей точкой (64 разряда), byte – байт – 8-разрядный код, none – множество значений, не содержащее ни одного элемента, string – обычные строки объектно-ориентированного программирования (C#).

Конструируемые типы:

Запись (record) – фиксированный набор типизированных полей. Есть поля 0, 1, ... n-1, каждое из которых представляет значение заданного в определении записи типа. Количество

полей – фиксировано, типы полей – фиксированы. Иногда полям сопоставляют имена, являющиеся идентификаторами.

Последовательность (sequence) – упорядоченный набор, состоящий из неопределенного (ноль или более), но конкретного, числа однотипных элементов.

Объединение (union) – значение, состоящее из тега и подзначения. Тег (динамически) определяет вариант типа для подзначения. Теги нумеруются, начиная с 0.

Объектное представление. Любое структурное значение может быть представлено в виде объекта по следующей схеме: примитивные типы реализуются значениями соответствующих системных типов: логического, целого, длинного целого, с плавающей точкой (double), символьного (char), байта, строки. Значения типа none представляются значениями null. Конструируемые типы представляются массивами object[], элементами которых будут подзначения структур. Запись – массив объектных значений своих элементов, последовательность – массив объектных значений элементов. Значение объединенного типа представляет собой массив из двух элементов. Первый элемент – целое значение тега. Второй элемент – объектное представление подзначения соответствующего тегу типа.

Примеры. Объектные значения (object)22, (object)"demo string", (object)true могут быть проинтерпретированы только как целое, строка и логическое. Объект new object[] {2, 33} может быть проинтерпретирован в зависимости от типа или как запись двух целых или как последовательность целых или как объединение с вариантом 2 и подзначением 33. Для последовательности записей, состоящих из строкового и целого, значением в объектном представлении может быть: new object { new object[] {"str1", 111}, new object[] {"str2", 222} }

Текстовая сериализация. Атомарные значения изображаются в виде текста: типа none – пустой строкой, типа boolean, character, integer, longinteger, real – традиционно, как напр. в C#. @byte изображается 16-ричным кодом, строка – как обычно (в C#). Запись изображается заключенным в фигурные скобки перечислением через запятую значений всех полей записи, начиная с нулевого и далее по порядку. Возможно использование имен полей (как в Паскале). Последовательность, изображается перечислением элементов через запятую, все перечисление (нуль или более элементов) помещается в квадратные скобки. Объединение изображается тегом – числом в диапазоне 0-255, следующим за ним символом ^ и далее идет изображение значения того типа, который динамически задан тегом. Значение может быть помещено в круглые скобки, это нужно для определенности разбора. По структурному значению и его типу однозначно, с точности до несущественных синтаксических элементов (пробелы, перевод строки, tab) и "лишних" скобок, определяется текстовая развертка. По корректной текстовой развертке и типу, однозначно определяется структурное значение.

Примеры. Для структурного значения типа последовательности записей строкового и целого полей, заданного в объектном виде как new object { new object[] {"str1", 111}, new object[] {"str2", 222} }, текстовая сериализация будет: [{"str1", 111}, {"str2", 222}].

Бинарная сериализация. Бинарная сериализация выполняется следующим образом: атомарные значения отображаются на последовательность байтов так, как определяется системой программирования .NET (C#) через метод System.IO.BinaryWriter и двойственный ему System.IO.BinaryReader. Байт отображается в байт, целое в 4 байта, длинное целое - в 8 байтов, число с плавающей точкой в 8 байтов, значение типа none - в 0 байтов. Строка отображается сложнее. Сначала идет целое (1 или более байтов) число, равное количеству символов в строке. Далее, если количество символов больше 0, то идут символы в виде потока байтов, сформированных по правилам UTF-8: byte[] info = new UTF8Encoding(true).GetBytes(str). Запись сериализуется поставленными "встык" сериализованными значениями полей. Последовательность вначале имеет 64-разрядное (8 байтов) целое, фиксирующее количество элементов последовательности (0+), а за ним подряд идут соответствующее количество значений элементов последовательности. Объединение реализуется постановкой 1 байта, который фиксирует код тега (0-255), за этим байтов непосредственно следует подзначение соответствующего тегу типа.

Язык определения типов. Поляровские типы задаются некоторыми формулами, задающими типовые (ударение на первый слог) значения. Типовые значения обычно создаются специальными конструкторами классов PType, PTypeRecord, PTypeSequense, PTypeUnion. Кроме того, эти значения можно рассматривать как обычные структурные значения и существуют процедуры перевода из поляровского объектного представления в типовое и обратно. Также существует (как и в других языках) специальный язык определения типов. Синтаксис языка в БНФ:

```
типовое_определение: имя_типа = типовая_формула ;
типовая_формула: none
     | bool | byte | char | int | long | float
     | { (имя поля : )? типовая_формула,.. }
     | [ типовая_формула ]
     | имя_тега ^ типовая формула
```

3. Общая часть библиотеки программ

Есть базовая часть рассматриваемых решений – пространство имен Polar.DB. В сборке размещен ряд ключевых компонентов, применяемых в других пакетах библиотек. Наиболее принципиальные компоненты связаны с типами. Как уже отмечалось, поляровское структурное значение осмыслено в контексте его типа. А тип, это также структурное

значение, формируемое или статически, как константа в коде, или динамически, с помощью подходящих средств. Отметим рекурсивность ситуации с типами: значение осмысленно только в контексте другого значения. Соответственно, у типового значения также есть тип, в библиотеке он задается статическим значением TType.

Объекты поляровского типа имеет класс PType, конструктор этого класса позволяет конструировать стандартные простые типы: булевское, целое, вещественное, строка и др. Составные типы создаются на базе классов PTypeRecord, PTypeSequence, PTypeUnion, формирующих типы записи, последовательности и объединения соответственно. Кроме того, есть возможность превращать типовые значения в объекты и наоборот – объекты трансформировать в типовые значения. Такая гибкость дает средства комбинирования типов и значений. Например, можно записать в файл сначала типовое значение, а потом значение этого типа. Соответственно, при чтении, корректной будет последовательность чтения объекта типа (на базе TType) с переводом его в тип, напр. T, с последующим чтением объекта (на базе T).

В целом, тип может быть создан, у него есть признаки (Properties) и атрибуты, но главное – он может быть использован при работе со структурированными данными. Как уже указывалось ранее, поляровские структурированные данные имеют объектное представление (в оперативной памяти) и два вида разверток – текстовую и байтовую. Текстовая развертка – это текстовое представление структурного значения. Оно несколько похоже на JSON [6], однако не является самодостаточным. Это означает, что для правильной интерпретации текстового представления поляровского значения все равно требуется явно указанный тип, хотя в простых случаях, текст легок для понимания. Есть ряд базовых процедур, затрагивающих «тройку» тип, объект, среда развертки в процедурах сериализации/десериализации:

```
public static void TextFlow.Serialize(TextWriter tw, object v, PType tp);
public static object TextFlow.Deserialize(TextReader tr, PType tp);
public static void ByteFlow.Serialize(BinaryWriter bw, object v, PType tp);
public static object ByteFlow.Deserialize(BinaryReader br, PType tp);
```

Эти процедуры или формируют поток символов/байтов из объектного представления значения (сериализация) или создают объект значения по имеющемуся потоку (десериализация).

В общей части библиотеки PolarDB также имеется поддержка для основной структуры для многих построений – последовательности. В частности, есть поддержка последовательностей с нефиксированным размером элементов и с фиксированным размером, есть добавление элементов, «очищение» последовательности, выстраивание индексов, использование шкалы.

Эта часть библиотеки предназначена для программирования самых быстрых решений по обработке, в которых особенности структуризации, хранения и доступа, добавляются программно, а не за счет универсализма элементов и действий.

4. Поляровские ячейки и индексы

Некоторая фиксация ряда решений, связанных с отображением структур на потоки байтов (Streams), делается в пакетах Polar.Cells и Polar.CellIndexes. Ячейки класса PaCell – это специально организованные байтовые потоки, как правило – файлы, для хранения структурированных данных и манипулирования этими данными. Естественно, каждая ячейка предназначена для хранения данных только одного типа. Ячейку можно создать или, если уже существует, подсоединиться. В ячейку можно положить значение предписанного типа или прочитать значение. Можно сказать, что ячейка – «внешняя» память для структурированных значений. Эта память может быть уже занятой значением и тогда другое значение не запишется, но есть явный метод очищения (Clear()) ячейки, тогда можно снова записывать.

К динамически изменяемым базам данных ячейка имеет то отношение, что последовательность верхнего уровня может не только записываться целиком, но и расти через последовательное добавление элементов. Для считывания из ячейки не всего значения, а части, существует механизм выделения части с последующим чтением. Это делается через некоторый аналог указателей – структуру класса PaEntry. Технология здесь следующая. Значение класса PaEntry указывает на часть общего значения – на подзначение. Это подзначение обладает каким-то типом и этот тип непосредственно динамически зафиксирован в указателе. А далее, есть набор методов «углубления» в подзначение – выделение поля для записи, выделение элемента в последовательности и выделение варианта в объединении. А в конце углублений значение можно прочитать. Или даже записать вместо имеющегося в поле содержимого, новое значение. Но это допустимо только для полей фиксированного размера, таких как числа, байты и записи, состоящие из подзначений фиксированного размера. Фиксированный размер определен для типа или он зависит от значения. Например, числа являются значениями фиксированного размера, а строки – не являются.

В набор методов работы с полями также входят сканирования элементов последовательности, когда вырабатывается поток значений имеющихся значений или для каждого элемента выполняется заданное обработчиком действие.

В пространстве имен Polar.CellIndexes библиотеки расположено конкретное индексное решение, точнее некоторое семейство решений. Суть его заключается в том, что индексное построение для последовательности (в частном случае – таблицы) выстраивается «вокруг» класса TableView, который представляет собой последовательность элементов, заданного в определении поляровского типа, снабженных дополнительным логическим признаком deleted. Если этот признак истинен, то элемент считается уничтоженным. Эта конструкция позволяет выполнять полный набор редактирующих операций: добавление элемента, уничтожение элемента и изменение элемента (через уничтожение и добавление нового). Причем уничтожение элемента выполняется не «физическим изъятием», а отметкой в соответствующем поле.

Индексные построения выполняются по специальной схеме, когда формируется «статическая» часть индекса в виде сортируемой последовательности, и она расширяется динамической частью индекса, выполненной в виде структуры в оперативной памяти (применяется хеш-таблица Dictionary). Индексы создаются относительно независимо и регистрируются в опорной таблице. Соответственно, далее операции, требующие отработки в индексной структуре, автоматически доносятся до структуры через регистрацию. При создании индекса, задается функция вычисления ключа на значениях элементах последовательности. Также указывается таблица, к которой относится индекс и, возможно, шкала. Шкала – это отображение значений числового ключа или полуключа в диапазон отсортированного индексного массива в котором элементы с такими значениями могут находиться. Такое отображение делается через равномерное разделение множества значений ключа с фиксацией в отдельном массиве (шкале) получившихся диапазонов.

Достоинством такай схемы индексного построения, является высокая эффективность работы и использования оперативной памяти. Недостатком является необходимость периодического перестроения индекса, в том числе – в динамике обслуживания запросов. Недостаток не проявляется для типичных ситуаций применения динамических индексов. Статическая часть индекса (индексный массив) различается в зависимости от того, производится ли индексация по ключу фиксированного размера (IndexKeyImmutable), произвольному ключу (IndexViewImmutable) или полуключу, когда используется хеш-функция ключа (IndexHalfkeyImmutable). Причем для первого и третьего вариантов возможно использовать шкалу, что заметно ускоряет доступ к хранимым элементам.

В пространстве имен Polar.CellIndexes имеется также 2 сформированных решения для типичных случаев применения библиотеки. Первое – универсальная таблица имен (NameTableUniversal), второе – простая таблица (TableSimple). Суть таблицы имен

заключается в порождении биекции между множеством строковых значений и множеством числовых кодов. Причем ее организация оптимизирует по времени преобразование строка –> код и обратно код –> строка. Подобные построения востребованы в современных информационных технологиях, в том числе, связанных с большими данными, их место, в основном в реализации key-value хранилищ данных.

Простая таблица также формируется на основе универсальной таблицы и универсальных индексов раздела. Но для упрощения и задания таблицы, все делается в «стандартном» для парадигмы реляционных таблиц представлении, когда задаются столбцы и на столбцы «навешиваются» индексы. Соответственно, задание таблицы довольно лаконично и интуитивно понятно. Например, конструктор класса выглядит как:

```
    public TableSimple(PType tp_elem, int[] indexcolumnnoms, Func<Stream>
getstream);
```

В нем задается тип элементов, предполагается, что это запись (набор колонок), второй аргумент определяет номера индексированных колонок, функция задает генератор байтовых потоков (Streams), нужных для выполнения построения. Генератор потоков должен быть таким, чтобы при запуске конструктора простой таблицы, потоки бы создавались, а при уже созданных потоках, происходило бы подключение к ним.

5. Странично организованные потоки байтов

Пространство имен Polar.PagedStreams предназначено для реализации программными средствами произвольного набора потоков байтов (Streams). В качестве «строительного материала» для этого используются подходящие массивы байтов (блоки, страницы) одинакового размера. Данный раздел библиотеки позволяет решить множество задач, связанных с хранением данных и с доступом к данным. К таким задачам относятся: упаковка базы данных или СУБД в один файл или предписанное количество файлов, экономная работа с большим и очень большим (миллионы и более) числом потоков, использование специальных (своих) схем кеширования доступа к файловым объектам, построение универсальных и специализированных решений по обеспечению надежного хранения через задаваемую избыточность и схемы контроля и восстановления.

Общая схема логического устройства блочной или страничной памяти задается абстрактным классом SetOfBlocks. Пока в библиотеке имеется единственная реализация, расширяющая этот класс FileOfBlocks, использующая файл как источник блоков через простое «нарезание» потока байтов на логически независимые блоки. Соответственно, файловый набор блоков через методы предоставляет пользователям произвольное

количество объектов класса PagedStream, имеющий интерфейс Stream, будем говорить о FileofBlocks как о генераторе потоков.

Организация страничной памяти байтовых потоков похожа на классические решения в ОС Unix [7], в частности, дескриптор потока имеет вид:

```
struct {
      long stream_length, n_blocks;
      struct { long D0, D1, D2, D3, D4, D5, D6, D7, D8, D9, D10, D11, D12;}
sub_blocks;
      long beginblock_length;
}
```

При этом, поток начинается как продолжение дескриптора, потом 10 длинных целых указывают на начала следующих страниц, D10 указывает на следующие страницы с одинарной косвенностью, D11, D12 – с двойной и тройной соответственно.

Существенным отличием примененного подхода является то, что нет встроенной системы директорий, директорных и файловых атрибутов. Считается, что эти свойства могут быть привнесены в конкретной программе и для этого, библиотечных средств достаточно. Например, простое key-value файловое хранилище можно организовать из трех потоков. В одном потоке хранится последовательность записей, состоящих из ключа, метаинформации (включая имя файла), начала и длины хранимого файла. Во втором потоке – байты подряд идущих файлов, в третьем потоке – индекс по ключу первой последовательности.

Создание простого кеша страниц для хранилища потоков, еще одно свойство реализации FileOfBlocks. В принципе, это свойство необязательно и может быть отключено, но если используется, то это может заметно повысить скорость доступа к данным.

6. Реализация, применения, эффективность

Библиотека реализована в системе .NET Core на языке программирования C# в виде открытого кода, расположенного в репозитории https://github.com/agmarchuk/PolarDB. Также она доступна через Nuget-пакеты с именами Polar.DB, Polar.Cells, Polar.CellIndexes, Polar.PagedStreams.

Поскольку .NET Core – многоплатформенное решение, библиотека может быть применена для программ под Windows, Linux, iOS. Первые два варианта успешно были проверены.

Библиотека применяется в проектах по исторической фактографии, ведущихся в ИСИ СО РАН, НГУ. Она используется в проекте «Открытый архив СО РАН» - открытой платформы построения и интеграции электронных архивов.

Данная библиотека активно используется в обучении, в частности в учебном курсе «Алгоритмы и технологии работы с большими данными» https://github.com/agmarchuk/BDLearningCourse.

Библиотека рассчитана на простые формы работы с базой данных, когда всю работу можно вести через один модуль или сервис, когда синхронизация запросов может выполняться в рамках последовательного исполнения и когда не требуется организовывать сложные транзакции. В таких условиях, библиотека показывает хорошие свойства и высокие характеристики, к которым относятся: компактность, способность работать на устройствах с малыми ресурсами, высокая скорость загрузки данных, высокая скорость доступа к данным, отсутствие потребности в сервере.

Список литературы

1. Марчук А.Г., Лельчук Т.И. Язык программирования Поляр: описание, использование, реализация. Новосибирск, 1986. 96 с. [Marchuk A.G., Lel’chuk T.I. Yazyk programmirovaniya Polyar: opisanie, ispol’zovanie, realizatsiya, Novosibirsk, 1986, 96 s. (in Russian)].
2. 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.
3. Марчук А.Г. PolarDB - система создания специализированных NoSQL баз данных и СУБД // Моделирование и анализ информационных систем. Т. 21, № 6 (2014), с.169-175.
4. А.Г.Марчук, С.В.Лештаев Экспериментальная реализация Sparql-1.1 и RDF Triple Store // Аналитика и управление данными в областях с интенсивным использованием данных, XVII Международная конференция DAMDID/RCDL’2015, Обнинск, 13-16 октября 2015 года, Труды конференции, сс. 83-87.
5. 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.
6. Введение в JSON // https://www.json.org/json-ru.html
7. Карпов В.Е., Коньков К.А. Основы операционных систем. Курс лекций // Учебное пособие / Под редакцией В.П. Иванникова. — Электронное издание. — М.: ИНТУИТ. РУ "Интернет университет информационных технологий", 2005. — 536 с. — ISBN: 5-9556-0044-2