Документация к системе PolarDB

(в свободном стиле)

# Введение

Система PolarDB, по тексту часто просто Поляр, это библиотека решений по структуризации данных предназначенная для построения специализированных систем управления данными, баз данных, систем управления базами данных. Система разработана в ИСИ СО РАН. Разработка выполнена в среде .NET (в настоящее время, в .NET Core). Основным языком программирования является C#, он же является языком демонстрационных примеров.

# Структуры данных

Основой Поляра является специальная типизированная структуризация данных. Это означает, что любое значение Поляра имеет тип. Причем тип, как правило, статически определен, т.е. определен на стадии исходного текста программы. Существуют атомарные типы, совпадающие с традиционным набором: булевский, байт, символ, целое, длинное целое, вещественное. К этому добавляется тип строкового значения. Строка в Поляре соответствует представлению о строках в объектно-ориентированных языках программирования. То есть это набор Unicode-символов, рассматриваемый как единое целое. То есть без возможности прямого изменения элементов или длины строки.

Составные типы формируются через конструкторы: запись, последовательность и объединение. Запись – это фиксированный набор значений разного типа. Последовательность – это набор элементов (ноль или более) заданного типа. Объединение – это пара тег-подзначение. При этом, для каждого варианта (тега) определен свой тип подзначения.

Структурные значения могут существовать в разных формах и как значения в оперативной памяти и как файловые объекты или элементы файловых объектов и как текстовые формулы. Причем возможно использование разных представлений и для объектов в ОЗУ и для файловых (байтовых) и для текстовых форм.

Текстовое представление структурных значений

Текстовое представление атомарных значений, в основном совпадают с представлениями, принятыми в большинстве современных языков и систем программирования: true/false, 127, ‘Z’, 9999, 7.7e-7, “строка в юникоде”. Неопределенности в интерпретации целых чисел как байт или целое или длинное целое не возникает, поскольку тип значение всегда определен на момент его интерпретации.

Текстовое представление составных типов в основном соответствует представлению JSON, т.е фигурные скобки используются для изображения записей: { k1: value1, k2: value2 …}, прямоугольные скобки группируют последовательность элементов: [“abc”, “def”, “ghk”] . Отличительной особенностью Поляровской структуризации является наличие значений объединенного типа. Текстовое представление таких значений: tag^подзначение. Собственно, если нет объединенного типа, то все обычно и похоже на JSON, только ключи в записях изображаются без кавычек – кавычки в ключах JSON это дань динамической типизации.

В силу особенностей Поляровской типизации структурных значений, синтаксис их текстового представления может быть сокращен: идентификаторы ключей в записях могут быть опущены вместе с двоеточиями, теги в объединениях могут быть заменены на численные значения (0, 1,…).

Объектное представление структурных значений

Структурные значения в оперативной памяти представляются следующим образом: атомарные значения – в соответствии с системой программирования, т.е. булевскими, байтами, символами, числами, строками; записи и последовательности – в виде массивов (array) значений элементов; объединения – в виде массива из численного значения тега и значения подэлемента. Все массивы представляются в системе программирования как массивы объектов (object[] в C#).

Пример:

object[] team = new object[] {

new object[] { 7001, "Иванов", 20 },

new object[] { 7001, "Петров", 28 },

new object[] { 7001, "Сидоров", 17 }

};

Поляровские типы как объекты

Структурные значения в Поляре всегда интерпретируются только в контексте явно или неявно заданного типа. Явно заданный тип – это объект класса PType. Поляровский тип, как правило, конструируется в виде формулы, напр.:

PType tp\_persons = new PTypeSequence(

new PTypeRecord(

new NamedType("id", new PType(PTypeEnumeration.integer)),

new NamedType("name", new PType(PTypeEnumeration.sstring)),

new NamedType("age", new PType(PTypeEnumeration.real))));

Соответственно, конструктором элементарных типов является базовый конструктор

PType tp = new PType(PTypeEnumeration.xxx);

Где xxx – один из вариантов: none, boolean, @byte, character, integer, longinteger, real, sstring.

А конструкторы составных типов задают информацию о структуре и типах элементов:

var tp1 = new PTypeSequence(тип\_элементов);

var tp2 = new PTypeRecord(new NamedType(имя\_поля, тип\_поля), …);

var tp3 = new PTypeUnion(new NamedType(имя\_варианта, тип\_варианта), …);

Для типовых значений определена функция интерпретации структурного значения в объектном представлении в виде текста, напр.:

tp\_persons.Interpret(team);

Важно, чтобы структурное значение строго соответствовало типу.

Сериализации

Применяемое структурирование – древовидное. Сериализацией назовем перевод (дерева) объектного структурного значения в последовательность информационных единиц. Это может быть перевод в последовательность символов или перевод в последовательность байтов. В первом случае получается текстовая сериализация, во втором – бинарная. Обратное преобразование из последовательности информационных единиц в объектное представление структурного значения назовем десериализацией.

Модуль TextSerialize содержит статические методы

public static void Serialize(TextWriter tw, object v, PType tp);

public static object Deserialize(TextReader tr, PType tp);

Первый – пишет в поток символов, второй – читает из потока символов и формирует объектное структурное значение.

Текстовая сериализация в общем-то предназначена для человека. Когда человек хочет увидеть данные, когда он хочет изменить эти данные наиболее традиционным образом – средствами текстового редактирования. В отличие от этого, бинарная сериализация является «машинной» и предназначена для главного – временного (кратковременного или долговременного) хранения структурированной информации. Соответственно, ключевым здесь является возможность какое-то структурное значение записать в файл, а потом, когда понадобится – прочитать. Модуль BinarySerialize содержит парочку статических методов, позволяющих это сделать:

public static void Serialize(BinaryWriter bw, object v, PType tp);

public static object Deserialize(BinaryReader br, PType tp);

Сочетание файла бинарных сериализации/десериализации дает универсальную среду работы с объектами. Действительно, файл может быть установлен на конкретную позицию чтения/записи:

Filestream.Position = offset;

После этого, можно (организовав бинарный райтер) записать структурное значение и, когда оно понадобится, снова установить позицию и десериализовать это значение.

Универсальная последовательность

Класс универсальной последовательности эффективно реализует в виде файлового (Stream) построения частный, но очень важный вариант структурирования – последовательность элементов заданного типа. Соответственно, у конструктора класса 2 параметра: поляровский тип элемента и файл (Stream) реализации:

UniversalSequenceBase usequence = new UniversalSequenceBase(tp\_person, filestream);

Самые общие действия над последовательностью – очищение Clear(), добавление элемента AppendElement(element) и сброс буферов Flush(). С помощью этих средств, можно создать последовательность с носителем на байтах файла и заполнить ее элементами. Следующая группа действий: прочтение потока элементов и сканирование:

public IEnumerable<object> ElementValues();

public void Scan(Func<long, object, bool> handler);

Аргументами для функции сканер представляет офсет (позицию в файле) очередного считываемого элемента и его считанного объектного значения. Функция после обработки элемента возвращает истину, если можно продолжать сканирование и ложь, если сканирование надо завершить.

Следующая группа действий – локализация, чтение и запись элементов. Локализация элементов осуществляется прямо или косвенно через вычисление офсета. Элементы в файле располагаются один за другим, поэтому офсет элемента равен офсету предыдущего элемента плюс размер предыдущего элемента. Позиция элемента последовательности вычисляется совсем просто когда размер элемента фиксирован. На практике это означает, что в записи нет строковых значений, последовательностей и объединений. В остальных случаях позиция элемента берется из сохраненных значений. Как правило, из индексов. Вот набор методов доступа к элементам последовательности:

public long Count();

public long ElementOffset(long ind); //Работает только для элементов фиксированного разм.

public void SetElement(object v, long off); // --- || ---

public object GetElement(long off);

public object GetByIndex(long index); // --- || ---

Последняя группа методов – сортировки. Дело в том, что универсальная последовательность используется также для построения индексов. А индексы надо уметь упорядочивать (сортировать) для более экономного доступа.

public void Sort32(Func<object, int> keyFun);

public void Sort64(Func<object, long> keyFun);

Методы переставляют элементы в последовательности так, чтобы они были упорядочены по заданной ключевой функции. Функция вырабатывает или 32-рядное целое или 64-разрядное целое.

Последовательности и индексы к ним

Универсальная последовательность дает решение, которое можно рассматривать лишь как полуфабрикат. Реально, к последовательности элементов предъявляется существенно больше требований, чем имеется для UniversalSequenceBase. В частности, важен весь цикл жизни и использования последовательности. Мы исходим из того, что последовательность заполняется значениями, что происходят выборки элементов, изменение элементов, уничтожение элементов. Эффективная реализация такого набора требований – существенно более сложная задача и решается некоторым набором интерфейсов и классов.

Концептуальная схема «поддерживающей» или опорной последовательности. Основу реализации последовательности, естественно, составляет значение класса UniversalSequenceBase. К нему добавляются индексы, которые позволяют делать выборки элементов по какому-то признаку (ключу, похожему образцу и т.д.). Последовательность становится очень похожей на key-value хранилище, поскольку его интерфейс в некотором смысле упрощается:

public interface IBearing

{

void Clear();

void Flush();

void Load(IEnumerable<object> flow);

void Build();

void Refresh();

IEnumerable<object> ElementValues();

void Scan(Func<long, object, bool> handler);

object GetItem(long off);

long AddItem(object item);

void DeleteItem(long off);

IIndex[] Indexes { get; set; }

}

Интерфейс неявно определяет «жизненный цикл» опорной последовательности: создание, наполнение элементами, построение индексов, перечисление или сканирование, выборки, добавления и уничтожения отдельных элементов. Добавления и уничтожения отдельных элементов называются «слабой динамикой». Слабой, поскольку основная динамика (изменения) для последовательности осуществляется загрузкой элементов и вычислением индексов Build(). Соответственно, объем действий слабой динамики предполагается существенно меньшим чем в загрузочно-индексных.

Интерфейс опорной последовательности дополняется интерфейсом индексного построения:

public interface IIndex

{

void Clear();

void Build();

void Flush();

void Refresh();

void OnAddItem(object item, long off);

void OnDeleteItem(long off);

}

В нем имеется набор операций создания/сброса/освежения. Но еще должны быть определены обработчики событий добавления элемента и уничтожения элемента.

Таким образом, после построения (через конструктор) опорной последовательности, регистрации в нем индексов и заполнения данными, весь конгломерат последовательность-индексы будет «действовать» единым ансамблем на операциях Build, Refresh, AddItem, DeleteItem.

Опорная последовательность это не только создание-загрузка-построение-использование, но и создание-«освежение»-использование. Это так называемое подключение (connection). От базы данных можно «отключится», только надо сделать Flush() для опорной последовательности. Для подключения к последовательности в дальнейшем, нужно создать ее через конструктор вместе с прописанными в ней индексами и выполнить метод Refresh(), который «доделает» динамические части индексов. Кроме того, Refresh() позаботится о «разогреве» базы данных, иначе ее работа может быть очень медленной.

Реализовано несколько видов индекса. Есть «ключевые» индексы, когда на значениях элементов вычисляется с помощью заданной функции некоторый ключ, как правило однозначно идентифицирующий элемент и индексное построение помогает быстро выделять элемент по ключу-образцу. Класс IndexKey32Comp представляет такой индекс. Его конструктор:

public IndexKey32Comp(Func<Stream> streamGen, IBearing bearing,

Func<object, bool> applicable,

Func<object, int> hashFun, Comparer<object> comp);

Определяет функцию генерации рабочих стримов, опорную последовательность, функцию применимости, ключевую/хеш функцию с 32-разрядным целым результатом и компаратор для сравнения элементов в объектном представлении. На сочетании функции и компаратора можно реализовывать разные идеи поиска. Самая простая идея: ключевое поле и «нулевой» компаратор. Это для ситуаций, в которых есть целочисленное поле, идентифицирующее запись. Если идентифицирующее поле напр. строка, то хеш-функция строки станет первичным ключом поиска, а компаратор – вторичным. Функция применимости (applicable) разделяет элементы на те, к которым данный индекс применим и те, к котором не применим и не вычисляется.

Другой реализованный индекс – IndexView. Его особенность в том, что основой индексного построения является массив офсетов, отсортированных в соответствии с компаратором, заданным на элементах. Конструктор индекса:

public IndexView(Func<Stream> streamGen, IBearing bearing,

Func<object, bool> applicable, Comparer<object> comp\_d);

Здесь почти все параметры те же, что и для IndexKey32Comp и смысли их тот же.

Есть еще векторный индекс. Векторным он называется потому, что каждый элемент порождает не один индексный элемент, а несколько (вектор). Из-за этой особенности, ему не нужна функция applicable. Это потому, что для неприменимых элементов достаточно сформировать результат в виде пустого массива. В остальном, он очень близок IndexKey32Comp. Его конструктор:

public IndexKey32CompVector(Func<Stream> streamGen, IBearing bearing,

Func<object, IEnumerable<int>> keysFun, Comparer<object> comp);

Как и для IndexKey32Comp, указание компаратора как null, устранит вторичную сортировку.