BIGDATA

Лекция 5

ПЛАН

- Parquet: остатки
- Реляционные СУБД в Hadoop
- Колоночная модель

ТЕРМИНОЛОГИЯ PARQUET

- Три дополнительных понятия: Row Group, Column Chunk и Page
- Row Group - это элемент деления первого уровня
- Цель - попасть в гранулярность MapReduce
- Он не должен быть распределенным

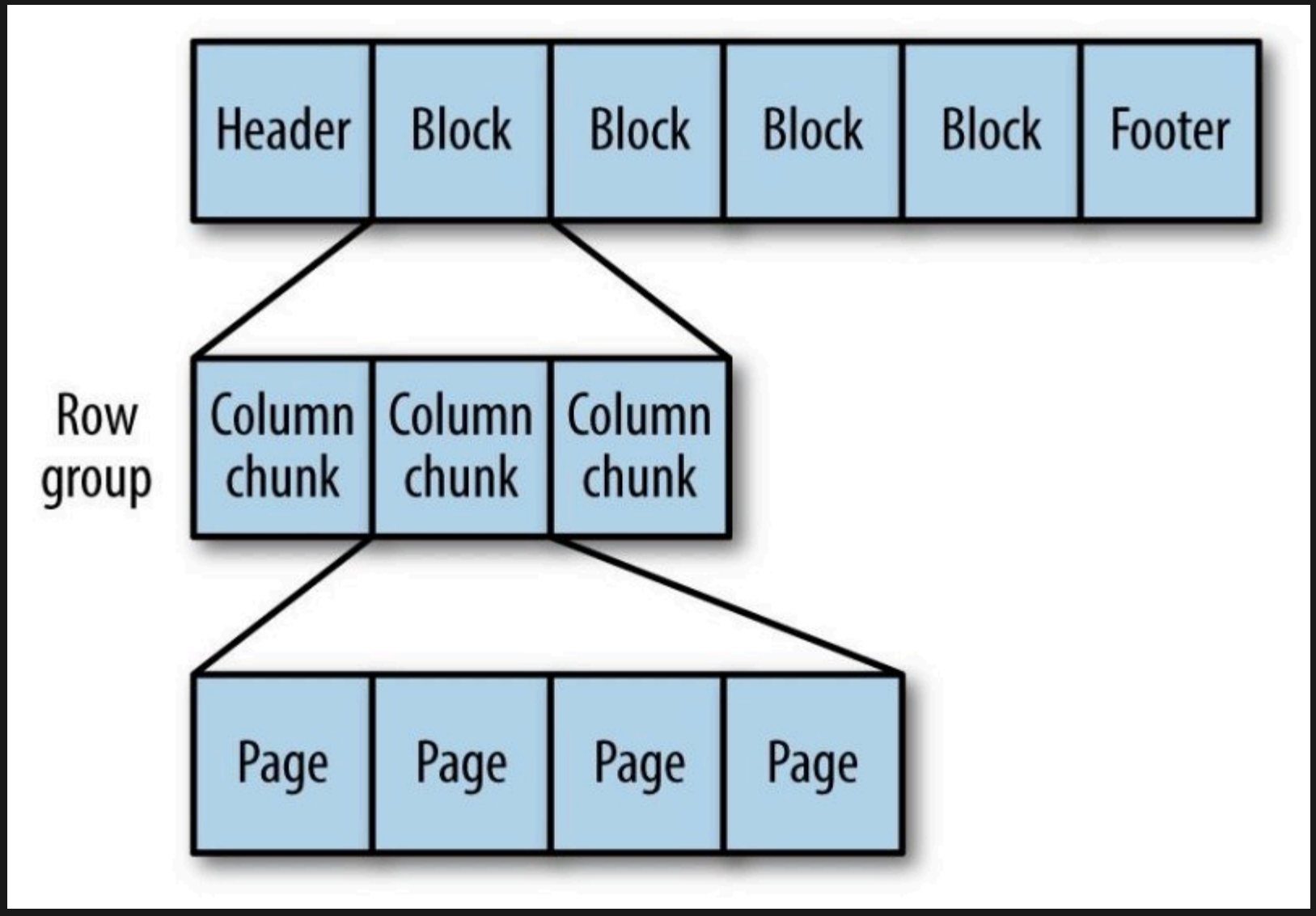
ТЕРМИНОЛОГИЯ PARQUET

- Column Chunk - транспонированная подтаблица
- Отображать на MR можно по-разному
- Можно передать полностью Column Chunk - но осторожно
- Можно передать по записям

ТЕРМИНОЛОГИЯ PARQUET

- Page - фрагмент колонки
- На этом уровне решается вопрос представления данных
- У каждой колонки по-своему
- Простейший вариант - просто рядом лежащие значения

Картинка


Header
Block
Block
Block
Block
Footer
Row group
Column chunk
Column chunk
Column chunk
Page
Page
Page
Page

ПРИМЕРНЫЕ РАЗМЕРЫ

- Row Group - 512M-1G
- С подстройкой HDFS-блока
- Data Page - 8K
- Чем меньше нужен точный поиск записи - тем больше стоить думать про увеличение Data Page

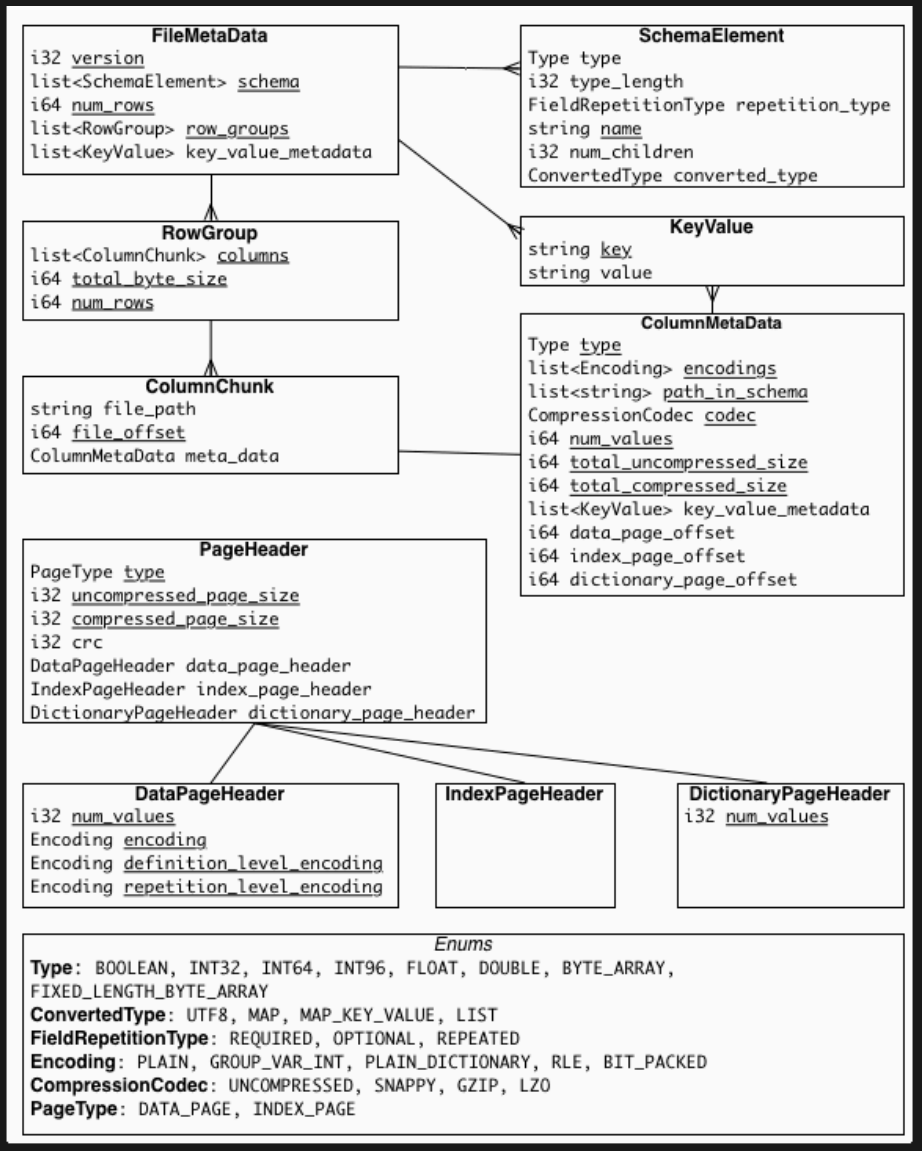
СХЕМА ДАННЫХ

- В метаданных хранится схема данных
- В простейшем виде - список колонок
- Но возможна иерархичность
- И повторяемость
- Разберем простой вариант

ПРОСТО ТАБЛИЦА

- Таблица с колонками базовых типов
- Базовые типы: boolean, int32, int64, int96
- float, double, byte_array, fixed_len_byte_array
- Есть понятие "логического типа"

Картинка


FileMetaData
i32 version
list<SchemaElement> schema
i64 num_rows
list<RowGroup> row_groups
list<KeyValue> key_value_metadata
SchemaElement
Type type
i32 type_length
FieldRepetitionType repetition_type
string name
i32 num_children
ConvertedType converted_type
RowGroup
list<ColumnChunk> columns
i64 total_byte_size
i64 num_rows
KeyValue
string key
string value
ColumnChunk
string file_path
i64 file_offset
ColumnMetaData meta_data
ColumnMetaData
Type type
list<Encoding> encodings
list<string> path_in_schema
CompressionCodec codec
i64 num_values
i64 total_uncompressed_size
i64 total_compressed_size
list<KeyValue> key_value_metadata
i64 data_page_offset
i64 index_page_offset
i64 dictionary_page_offset
PageHeader
PageType type
i32 uncompressed_page_size
i32 compressed_page_size
i32 crc
DataPageHeader data_page_header
IndexPageHeader index_page_header
DictionaryPageHeader dictionary_page_header
DataPageHeader
i32 num_values
Encoding encoding
Encoding definition_level_encoding
Encoding repetition_level_encoding
IndexPageHeader
DictionaryPageHeader
i32 num_values
Enums
Type: BOOLEAN, INT32, INT64, INT96, FLOAT, DOUBLE, BYTE_ARRAY, FIXED_LENGTH_BYTE_ARRAY
ConvertedType: UTF8, MAP, MAP_KEY_VALUE, LIST
FieldRepetitionType: REQUIRED, OPTIONAL, REPEATED
Encoding: PLAIN, GROUP_VAR_INT, PLAIN_DICTIONARY, RLE, BIT_PACKED
CompressionCodec: UNCOMPRESSED, SNAPPY, GZIP, LZO
PageType: DATA_PAGE, INDEX_PAGE

ПОЛЕЗНЫЕ ССЫЛКИ

- Thriff-описание структуры parquet-файла
- (Не "источник истины")
- Словесное описание
- (крайне лаконичное)

РЕЛЯЦИОННЫЕ СУБД

- SQL-базис научились выражать
- И многое сверх него
- Но как-то не хочется на все случаи писать заново MR-задания
- Хочется писать SQL-запросы
- И чтобы кто-то эти задания породил по запросу

HIVE

- Нет своего внутреннего формата хранения
- Работает непосредственно с HDFS-файлами
- Может забрать его к себе
- Через переименование
- Или вообще обрабатывать файл в произвольном месте

SCHEME ON READ/WRITE

- Классический SQL - scheme on write
- При записи проверяем данные по схеме
- Hive - scheme on write
- Проверяем схему при чтении
- (Записи вообще может не быть)

КОЛОНОЧНАЯ МОДЕЛЬ

- Мотивирующий пример: данные о посещении пользователями товаров
- Мы умеем собирать лог событий
- Умеем проектировать MapReduce-задачи
- И SQL-like запросы через Hive
- Так что же нас не устраивает ?

ЧЕГО ХОТЕЛОСЬ БЫ

- Хочется выполнять быстрые запросы
- Более точечные, чем MR по всем данным
- И быстро видеть изменения
- Быстрее, чем раз в сутки или раз в неделю

НАЧИНАЕТСЯ NOSQL

- Представим себе немного странную таблице
- Первая колонка - ID пользователя
- А дальше - очень много колонок
- Ну прямо очень много - от слова "совсем"

НАЧИНАЕТСЯ NOSQL

- Для каждой страницы - по колонке
- Редко в какой есть хоть какое-то значение
- А могут быть колонки про заказы
- С именами типа '2024.10.05:laptop'
- В непустых значениях - детали посещения или заказа

ЗАЧЕМ ЭТО ВСЕ ?

- Утверждается, что можно придумать способ быстро обновлять данные
- В базовом варианте - устанавливать значение для данного пользователя и колонки
- И это будет легко исполнимо для просто установки значения
- Для обновления уже установленного - посложнее
- Но в каких-то случаях - тоже можно

ЗАЧЕМ ЭТО ВСЕ ?

- И быстро выполнить некоторые запросы на чтение
- Например, найти заданную колонку у заданного пользователя
- Или все непустые колонки у заданного пользователя
- Или пользователей в заданном диапазоне ключей

А ДРУГИЕ ЗАПРОСЫ ?

- Например, искать пользователей на заданной странице
- Или заказы по месту доставки
- А как хранить данные пользователя, кроме ID ?

ОТВЕЧАЕМ

- Проще всего - с последним
- Заведем дополнительные колонки
- И эти колонки будут скорее всего плотнее заполнены
- Но возможны варианты: тонко проработанные колонки на каждый атрибут или одна с протобуфом
- А от чего зависит выбор варианта ?

ОТВЕЧАЕМ

- От ожидаемых запросов
- Но это же неправильно ? Неправильно. Зато быстро
- А что с запросами по id страницы ?
- Создайте для них свою таблицу
- Но это дубликация данных ?

ОТВЕЧАЕМ

- Да, это дубликация данных
- А транзакции хотя бы есть ?
- Если очень частичный ACID внутри таблицы
- А как тогда с согласованностью таблиц ?

ОТВЕЧАЕМ

- Вариант 1: никак
- В аналитике полная точность не строго обязательна
- Вариант 2: отслеживать/оценивать масштабы нестыковок
- Отдельными процедурами добиваться соглосованности

ТЕРМИНОЛОГИЯ HBASE

- namespace - набор логически взаимосвязанных таблиц
- Аналог "database" в postgres
- table - набор строк с разреженным набором колонок
- Схемы нет в принципе

ТЕРМИНОЛОГИЯ HBASE

- Простая HBase-таблица может напоминать SQL-таблицу
- Но чаще это более сложная сущность
- Может напоминать набор SQL-таблиц
- Одна из которых описывает объект реального мира
- А другие - факты про него

ТЕРМИНОЛОГИЯ HBASE

- Column Family - набор колонок
- Логически больше похоже на SQL таблицу, чем HBase-таблица
- Но не один в один
- Физически - колонки одной строки и одной CF будет физически храниться рядом
- Что не факт для колонок просто одной строки

ТЕРМИНОЛОГИЯ HBASE

- Column qualifier - имя колонки внутри column family
- Набор column family - свойство таблицы
- Их обычно немного - редко когда двузначное число
- Column qualifier - могут отличаться у разных строк

ТЕРМИНОЛОГИЯ HBASE

- Cell - "ячейка" таблицы
- Определяется ключом строки, CF и CQ
- Содержит значение и timestamp
- Можно хранить несколько версий колонки

ОСНОВНЫЕ ОПЕРАЦИИ

- Создание таблицы: указывается имя таблицы и перечень CF
- put: установить значение колонки в строке
- get: узнать значение заданной колонки в заданной строке
- Вариант: все колонки строки (может быть заметно дороже)
- get: можно ограничивать CF

ОСНОВНЫЕ ОПЕРАЦИИ

- scan: пройтись по диапазону колонок
- Можно уточнить диапазон колонок
- Ограничить набор колонок

КАК ОНО РАБОТАЕТ

- Работает поверх HDFS
- В чем-то повторяет распределение ролей
- HMaster - аналог NameNode
- RegionServer - аналог DataNode

Картинка

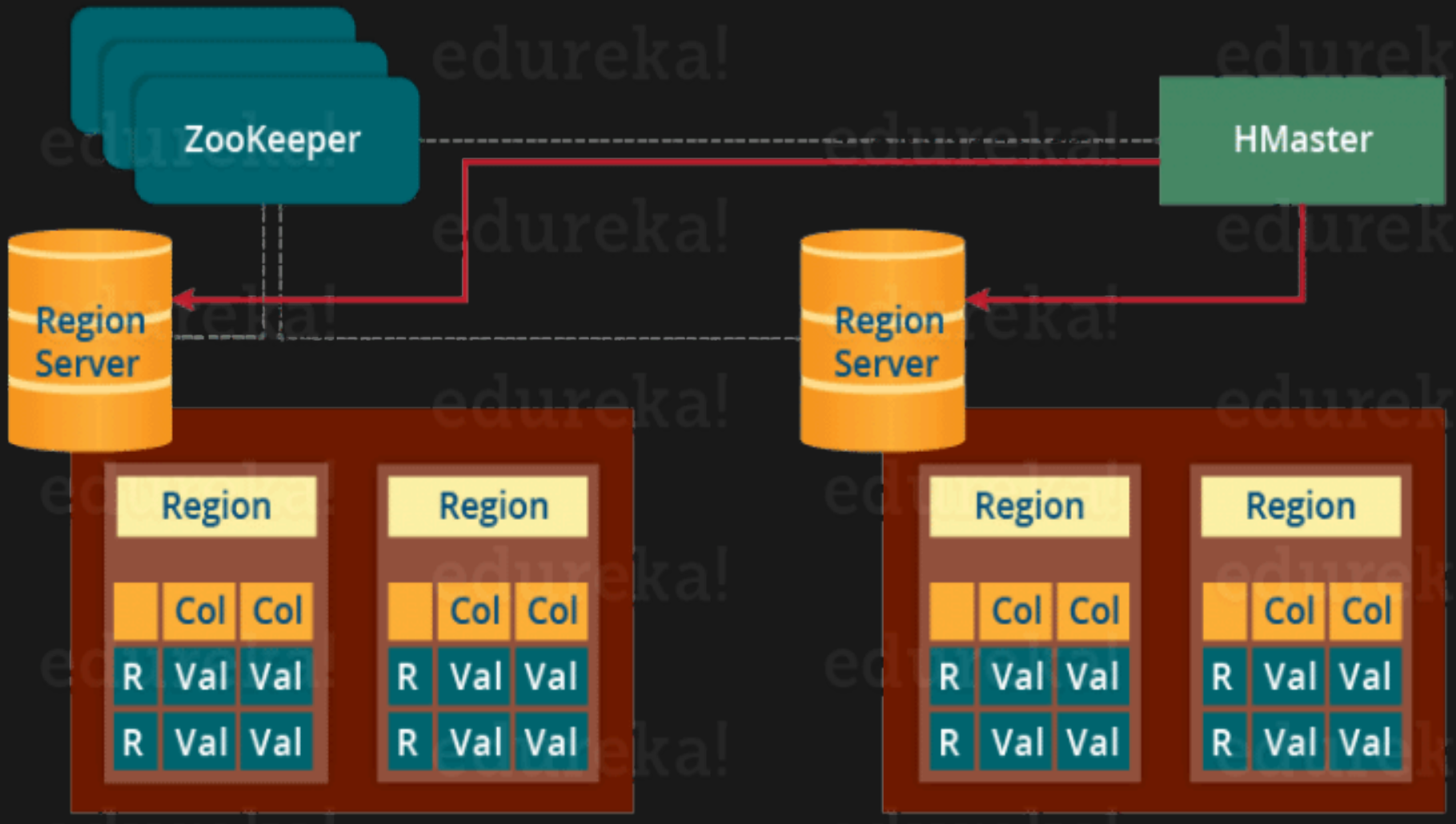


Figure: Components of HBase

КАК ОНО РАБОТАЕТ

- Таблица партиционируется по ключу строки
- Диапазон ключей назначается своему RegionServer-у
- RegionServer - это процесс, запускаемый на физическом узле
- В принципе - можно на любом, осмысленно - на HDFS-узле

КАК ОНО РАБОТАЕТ

- Пустая таблица сначала живет на одном RegionServer-е
- По мере роста происходит разбиение на два и выделение нового RegionServer-а
- HMaster хранит таблицу META
- В ней хранится знание о диапазонах ключей, назначенных RegionServer-ам

Картинка

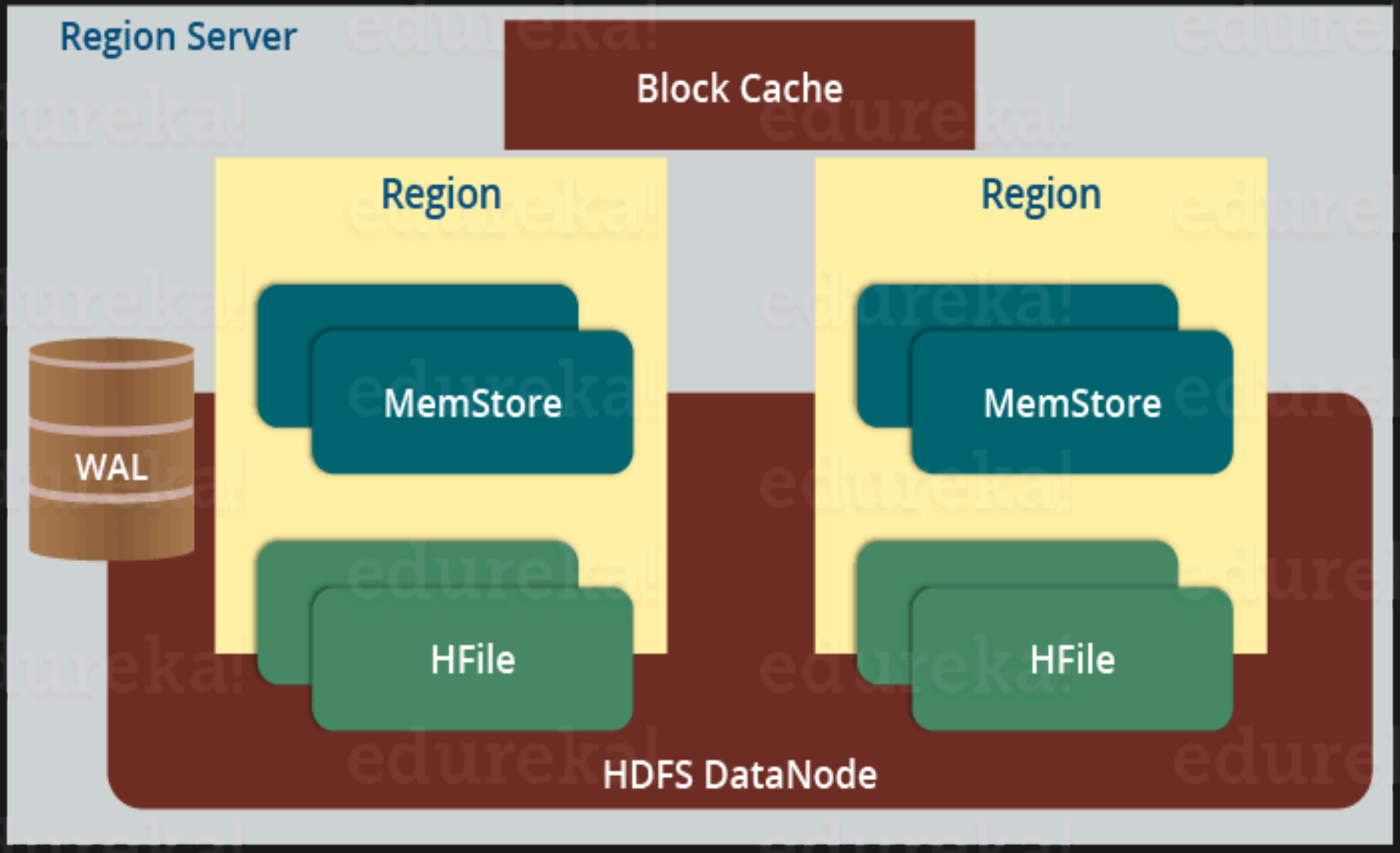


Figure: Region Server Components

КАК ОНО РАБОТАЕТ

- WAL - write-ahead log
- Помогает восстанавливаться при выключении
- Разные степени использования - настраивается на уровне таблицы
- Memstore - структура в памяти, что-то вроде дерева поиска

КАК ОНО РАБОТАЕТ

- Каждому CF соответствует один Memstore
- И много HFile-ов
- После записи в MemStore клиенту сообщается об успешной записи
- Когда HStore становится большим - записывается в новый HFile

ПРОМЕЖУТОЧНЫЙ ИТОГ

- Прорисовывается схема работы get/scan
- Беспокоит хранение неактуальных копий значения ячейки
- И рост числа файлов
- И в целом монотонность роста
- Мы вообще удалять что-то собираемся ? И как ?