

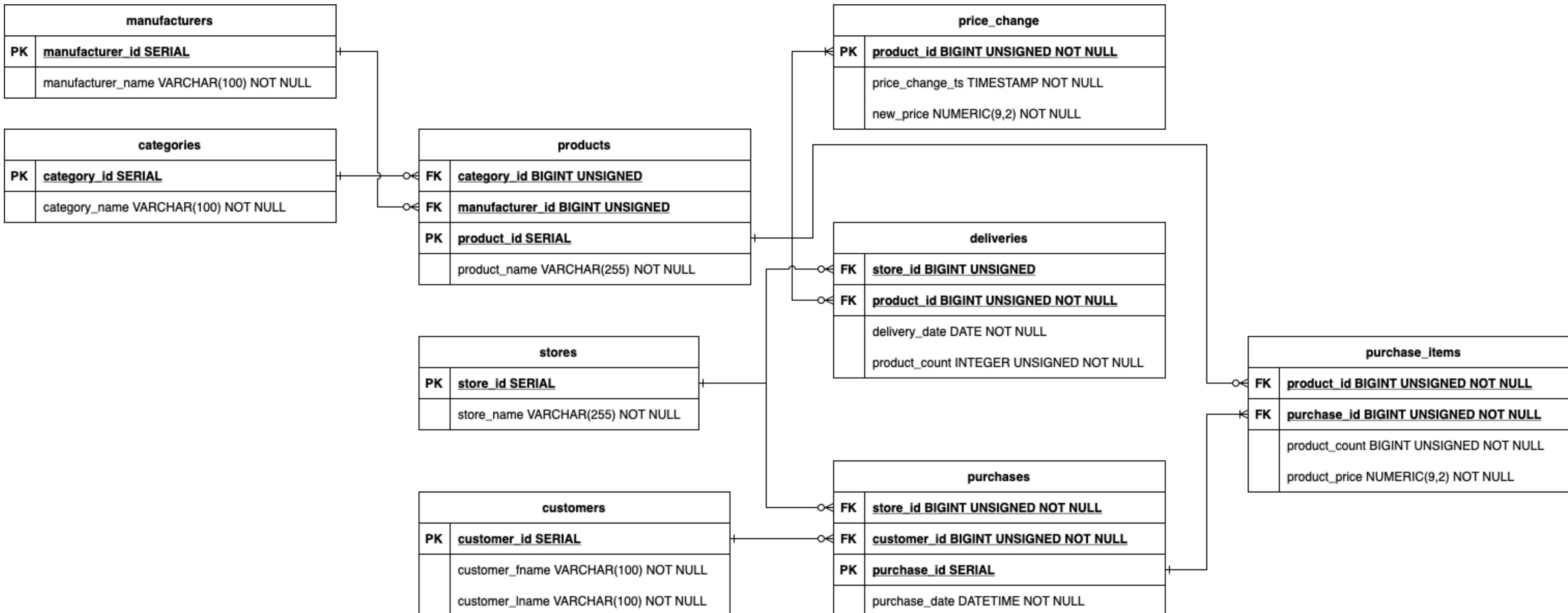
# **Modern Storages and Data Warehousing**

## **Week 3 - Column storages**

Попов Илья, [i.popov@hse.ru](mailto:i.popov@hse.ru)

# **1 - Домашнее задание 1**

# Домашнее задание 1



# Домашнее задание 1

- › Задача:
  - Написать для этой ER-диаграммы DDL;**
  - Запустить PostgreSQL в Docker,**
  - настроить создание БД из DDL из прошлого файла;**
  - Настроить репликацию;**
- › Максимальный балл за задание - 12
  - › Мягкий дедлайн - **15.10.2023 включительно**
  - › Дедлайн на 75% - **до дедлайна следующего ДЗ**
  - › Дедлайн на 50% - **до конца курса**
- › Бонусы:
  - Собрать скрипт для подсчета GMV;**
  - Собрать view на основе этого скрипта;**
  - Сделать так, чтобы ваш проект полностью собирался одной командой docker-compose up**
- › Готовое задание нужно загрузить на GitHub, далее - отправить его в Google Forms

**2 - В предыдущих сериях**

# Recap прошлых занятий

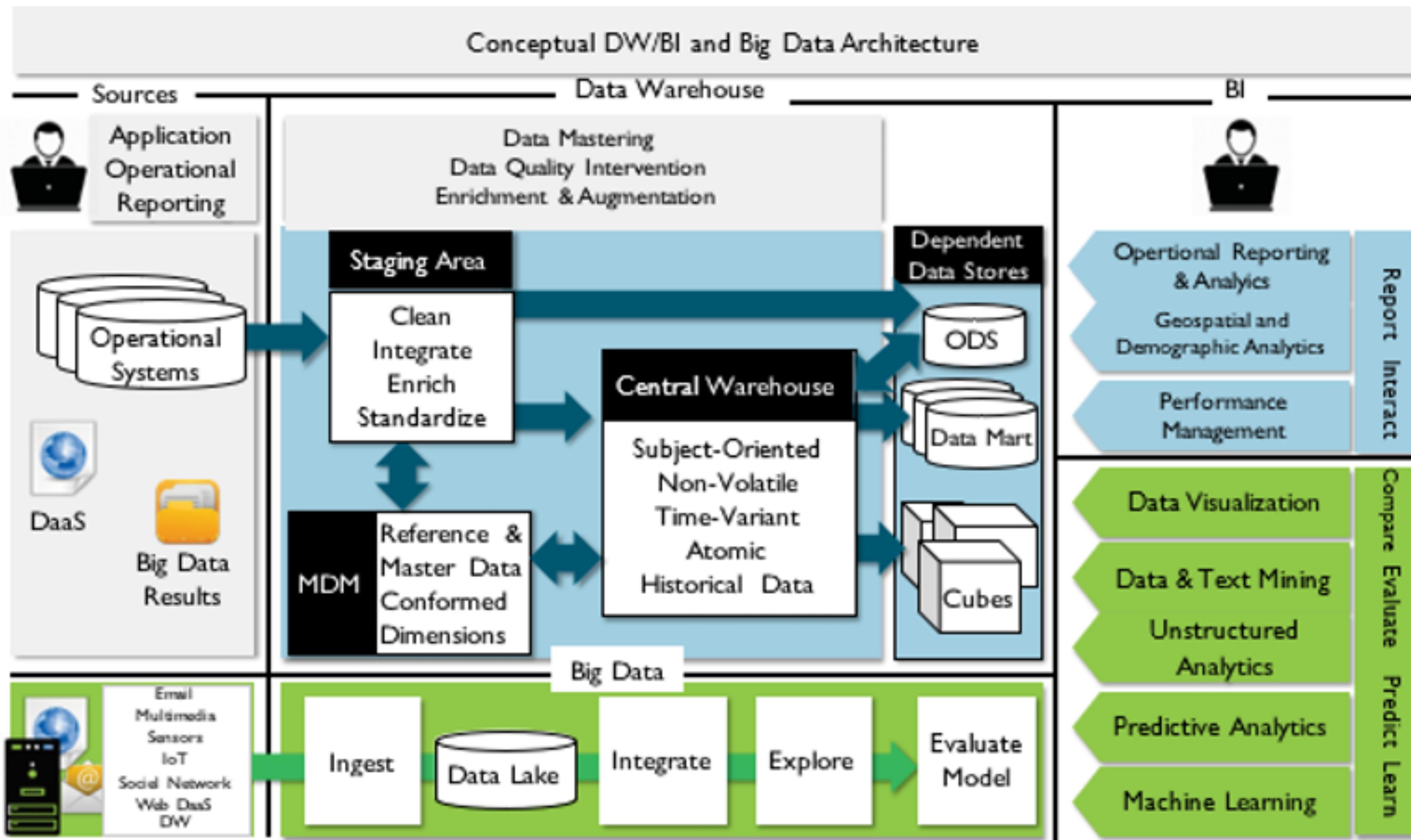
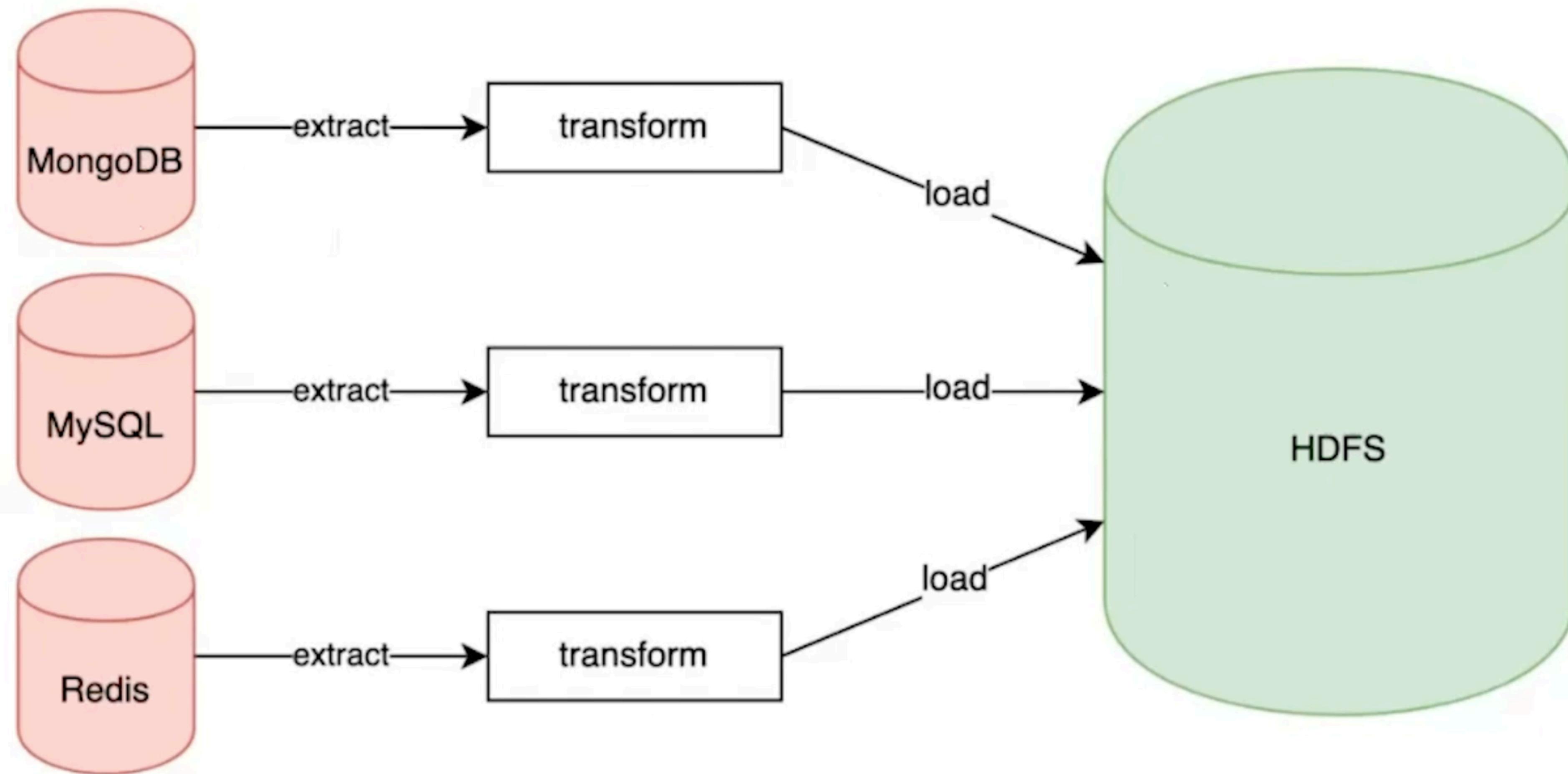


Figure 5: Date Warehouse Concept

# Recap прошлых занятий

- › Мы знаем, где рождаются данные (**OLTP**), почему проблемы **нельзя решить только масштабированием** и как его не убить при работе (**репликация**)
- › Мы знаем, куда сложить данные надолго и дешево (**S3**)
- › Мы знаем, куда сложить большой объем данных для последующей обработки (**HDFS**), как работает Hadoop-экосистема и как нам потом все-таки эти данные обработать (**MapReduce, Spark**)

# То, что мы знаем сейчас, выглядит примерно так



# Ресар прошлых занятий

- › Мы знаем, где рождаются данные (**OLTP**), почему проблемы **нельзя решить только масштабированием** и как его не убить при работе (**репликация**)
- › Мы знаем, куда сложить данные надолго и дешево (**S3**)
- › Мы знаем, куда сложить большой объем данных для последующей обработки (**HDFS**), как работает Hadoop-экосистема и как нам потом все-таки эти данные обработать (**MapReduce, Spark**)

Проблема:

- › Эту штуку мы хотим ускорить
- › Данные с прода можем возить не только мы в удобном виде, но и внешние подрядчики в неудобном виде

# **3 - Форматы хранения данных**

# Самый простой кейс

```
<?xml version="1.0"?>
<catalog>
    <book id="bk101">
        <author>Gambardella, Matthew</author>
        <title>XML Developer's Guide</title>
        <genre>Computer</genre>
        <price>44.95</price>
        <publish_date>2000-10-01</publish_date>
        <description>An in-depth look at creating applications
            with XML.</description>
    </book>
    <book id="bk102">
        <author>Ralls, Kim</author>
        <title>Midnight Rain</title>
        <genre>Fantasy</genre>
        <price>5.95</price>
        <publish_date>2000-12-16</publish_date>
        <description>A former architect battles corporate zombies,
            an evil sorceress, and her own childhood to become queen
            of the world.</description>
    </book>
    ....
```

```
spark.read
    .option("rowTag", "book")
    .option("rootTag", "books")
    .format("com.databricks.spark.xml")
    .load(path)
    .withColumn("price", cast("price"))
    .agg(f.max("price"))
    .show()
```

# Самый простой кейс

```
<?xml version="1.0"?>
<catalog>
    <book id="bk101">
        <author>Gambardella, Matthew</author>
        <title>XML Developer's Guide</title>
        <genre>Computer</genre>
        <price>44.95</price>
        <publish_date>2000-10-01</publish_date>
        <description>An in-depth look at creating applications
            with XML.</description>
    </book>
    <book id="bk102">
        <author>Ralls, Kim</author>
        <title>Midnight Rain</title>
        <genre>Fantasy</genre>
        <price>5.95</price>
        <publish_date>2000-12-16</publish_date>
        <description>A former architect battles corporate zombies,
            an evil sorceress, and her own childhood to become queen
            of the world.</description>
    </book>
    ...

```

```
spark.read
    .option("rowTag", "book")
    .option("rootTag", "books")
    .format("com.databricks.spark.xml")
    .load(path)
    .withColumn("price", cast("price"))
    .agg(f.max("price"))
    .show()
```

**Мысли?**  
**Бро/не бро?**  
**Будем так делать?**

# Какие форматы бывают вообще?

- › Специфичные для языка:
  - python — Pickle**
  - Java — Serializable, Kryo**
  - Ruby — Mashral**

# Какие форматы бывают вообще?

- › Специфичные для языка:
  - python — Pickle**
  - Java — Serializable, Kryo**
  - Ruby — Mashral**

**Мысли?  
Бро/не бро?  
Будем так делать?**

# Какие форматы бывают вообще?

- › Специфичные для языка:
  - python — Pickle**
  - Java — Serializable, Kryo**
  - Ruby — Mashral**
- › Нельзя десериализовать в другом языке
- › Плохо с версионированием
- › Плохо с производительностью

# Какие форматы бывают вообще?

〉 Текстовые форматы:

**JSON**

**XML**

**CSV**

**Yaml**

# Какие форматы бывают вообще?

〉 Текстовые форматы:

**JSON**

**XML**

**CSV**

**Yaml**

**Мысли?  
Бро/не бро?  
Будем так делать?**

# Какие форматы бывают вообще?

- › Текстовые форматы:
  - JSON**
  - XML**
  - CSV**
  - Yaml**
- › Проблемы с типами данных (различия в записи int/long/float/string)
- › Проблемы с экранированием
- › Избыточность
- › Неэффективное представление

# Какие форматы бывают вообще?

› Текстовые форматы:

**JSON**

**XML**

**CSV**

**Yaml**

› Решение проблемы - давайте просто рядом положим схему данных



# Что такое схема данных?

```
{  
    "type": "record",  
    "namespace": "com.example",  
    "name": "Person",  
    "fields": [  
        { "name": "firstName", "type": "string" },  
        { "name": "lastName", "type": "string" },  
        { "name": "age", "type": "int" }  
    ]  
}
```

# Что такое схема данных?

```
{ "type": "record", "namespace": "com.example", "name": "Person", "fields": [ { "name": "firstName", "type": "string"}, { "name": "lastName", "type": "string"}, { "name": "age", "type": "int"} ] } { "type": "record", "namespace": "com.example", "name": "Person", "fields": [ { "name": "firstName", "type": "string"}, { "name": "lastName", "type": "string"}, { "name": "age", "type": "int"}, { "name": "interests", "type": {"type": "array", "items": "string"}}, { "name": "address", "type": { "type": "record", "name": "mailing_address", "fields": [ {"name": "street", "type": "string"}, {"name": "city", "type": "string"}, {"name": "country", "type": "string", "default": "NONE"}, {"name": "zip", "type": "string", "default": "NONE"} ]}, "default": {} ] }
```

# Эволюция схемы

- › Со временем мы захотим менять/изменять формат, как наши данные хранятся
- › Эволюция схемы - это процесс её изменения во времени
- › О чём нужно думать:
  - Обратная совместимость - новый код может прочитать старую схему**
  - Прямая совместимость - старый код может прочитать новую схему**
- › В разных форматах совместимость реализуется по-разному

# Эволюция схемы

- › Со временем мы захотим менять/изменять формат, как наши данные хранятся
- › Эволюция схемы - это процесс её изменения во времени
- › О чём нужно думать:
  - Обратная совместимость - новый код может прочитать старую схему**
  - Прямая совместимость - старый код может прочитать новую схему**

# Apache Avro

- › Релиз в 2009 как часть Hadoop-экосистемы
- › Поддержка Spark из коробки
- › Библиотеки для сериализации/десериализации на любом языке
- › Использует схему данных и прямую/обратную совместимость

# Пример

- › Пример для MessagePack - один из видов JSON с бинарным представлением
- › Стерилизованное сообщение:

```
{  "userName": "Martin",  "favoriteNumber": 1337,  "interests": [    "daydreaming", "hacking"  ]}
```
- › Занимает 66 байт

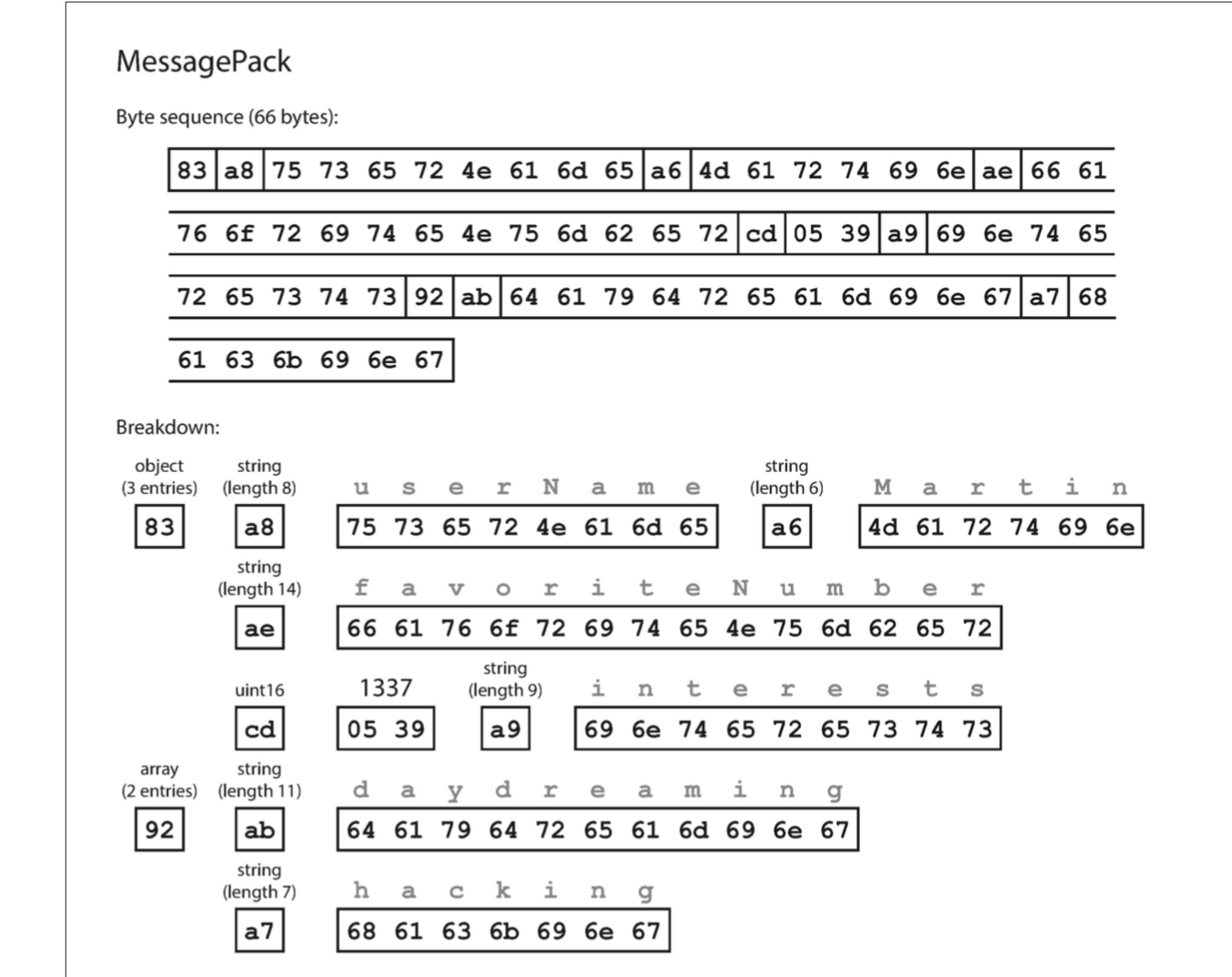


Figure 4-1. Example record (Example 4-1) encoded using MessagePack.

# Пример

- › Пример для Apache Avro
- › Стерилизованное сообщение:

```
{  
    "userName": "Martin",  
    "favoriteNumber": 1337,  
    "interests": [  
        "daydreaming", "hacking"  
    ]  
}
```
- › Занимает 32 байта

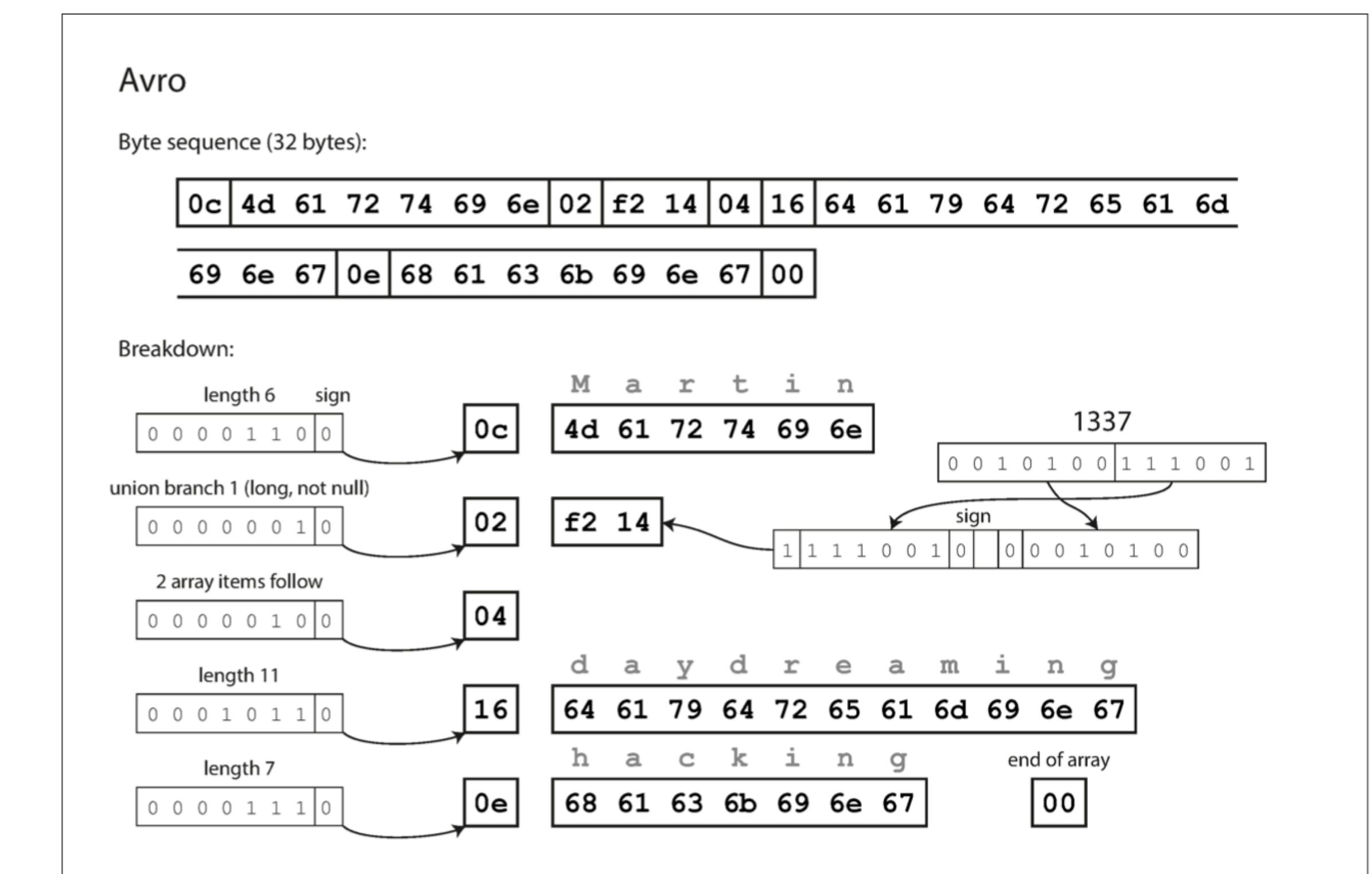


Figure 4-5. Example record encoded using Avro.

# Apache Avro

- › Схема хранится рядом с кодом в бинарном или json-формате
- › Схема хранится в данных в начале arvo-файла
- › Парсер смотрит на эти 2 схемы и понимает, как читать данные
  
- › Можно удалять колонки
- › Можно добавлять колонки (если задано default-значение)
- › Можно менять колонки местами
- › Можно менять тип данных, если типы конвертируются друг в друга



**Фиксировать схему данных -  
это хорошо**

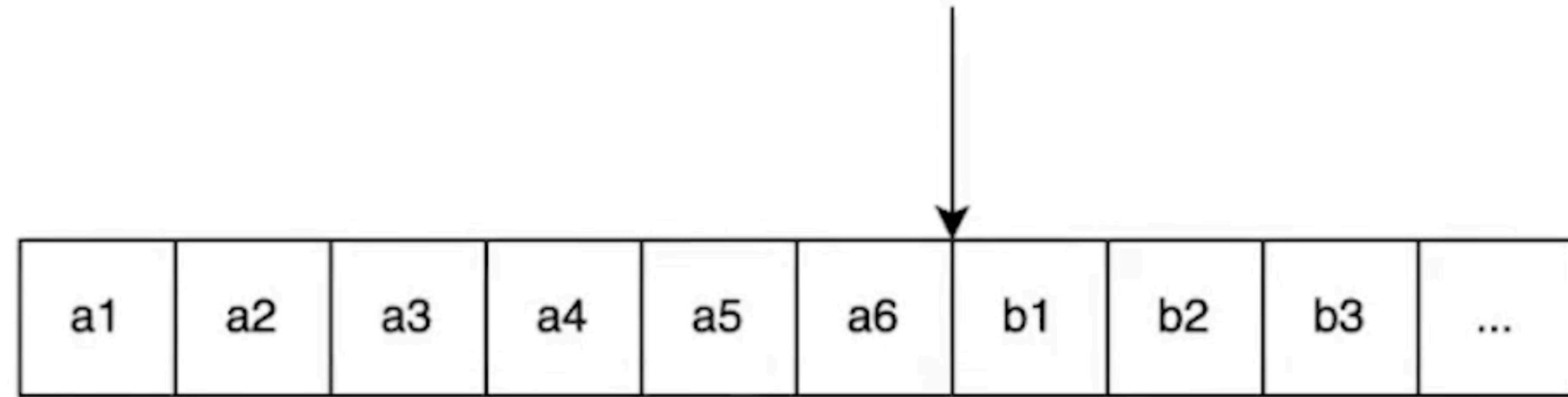
# Построчные форматы

	a	b	c	d
1	a1	b1	c1	d1
2	a2	b2	c2	d2
3	a3	b3	c3	d3
4	a4	b4	c4	d4
5	a5	b5	c5	d5
6	a6	b6	c6	d6

a1	b1	c1	d1	a2	b2	c2	d2	...
----	----	----	----	----	----	----	----	-----

# Поколоночные форматы

	a	b	c	d
1	a1	b1	c1	d1
2	a2	b2	c2	d2
3	a3	b3	c3	d3
4	a4	b4	c4	d4
5	a5	b5	c5	d5
6	a6	b6	c6	d6



# Поколоночные форматы

Плюсы:

- › Можно прочитать только одну или несколько колонок
- › Можно оптимизировать хранение - применить кодирование (dictionary encoding) или сжатие (RLE)
- › Можно использовать векторизированные вычисления

# Поколоночные форматы

Плюсы:

- › Можно прочитать только одну или несколько колонок
- › Можно оптимизировать хранение - применить кодирование (dictionary encoding) или сжатие (RLE)
- › Можно использовать векторизированные вычисления

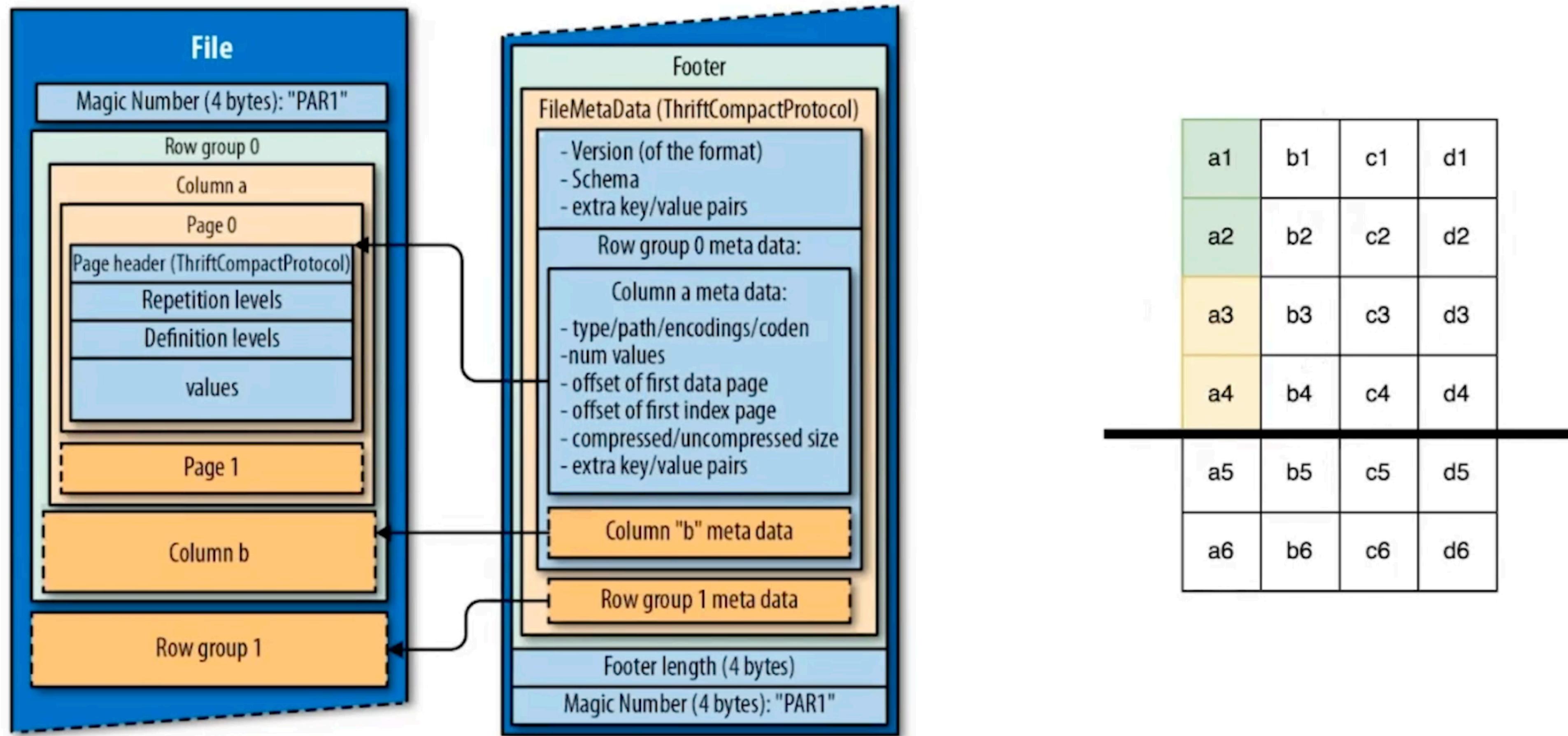
Важно:

- › Эффективность сжатия зависит от сортировки

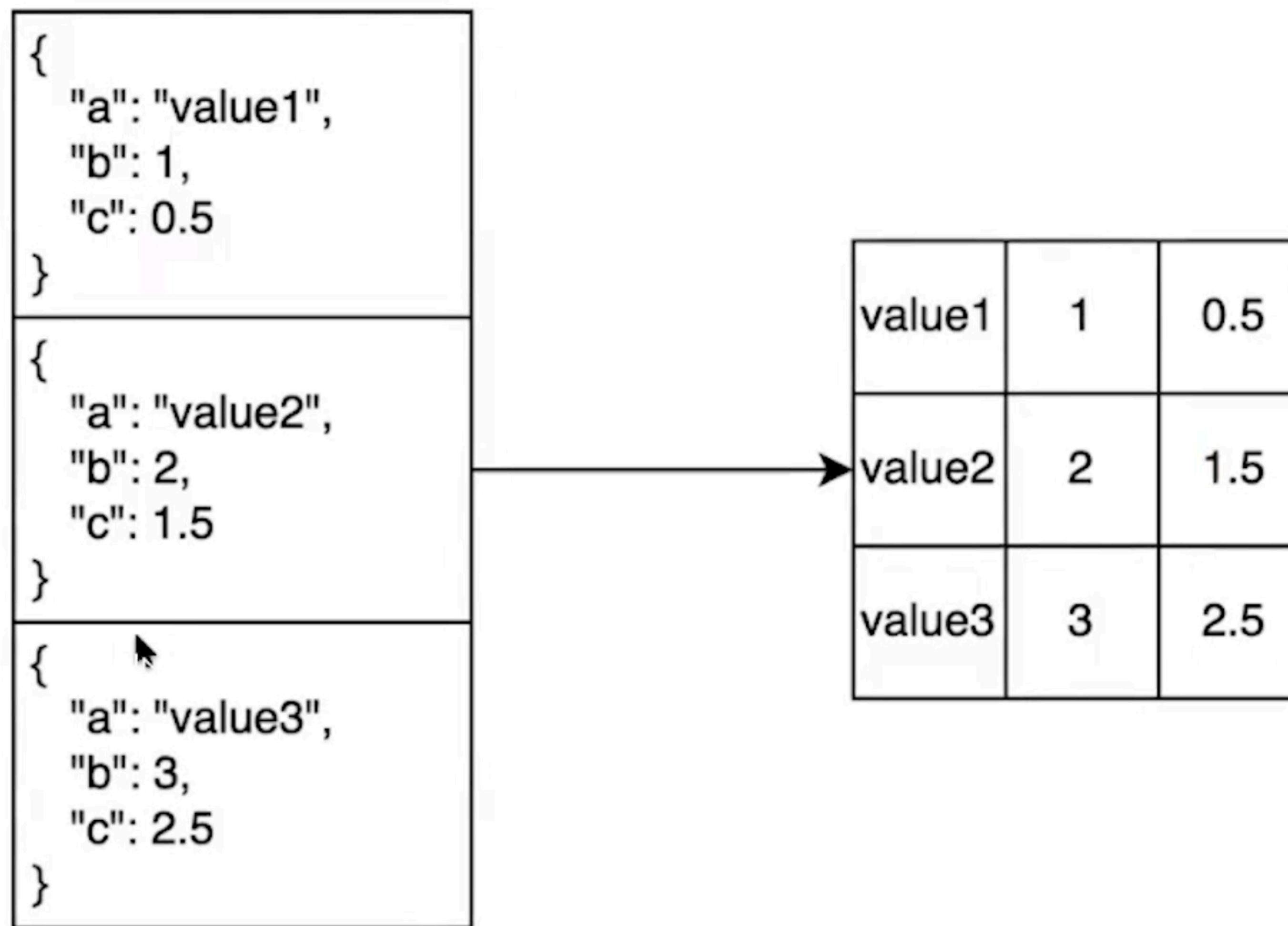
# Apache Parquet

- › Релиз в 2009 как часть Hadoop-экосистемы
- › Поддержка Spark из коробки
- › Библиотеки для сериализации/десериализации на любом языке
- › Использует схему данных и прямую/обратную совместимость
  
- › Использует метаданные и pushdown predicate

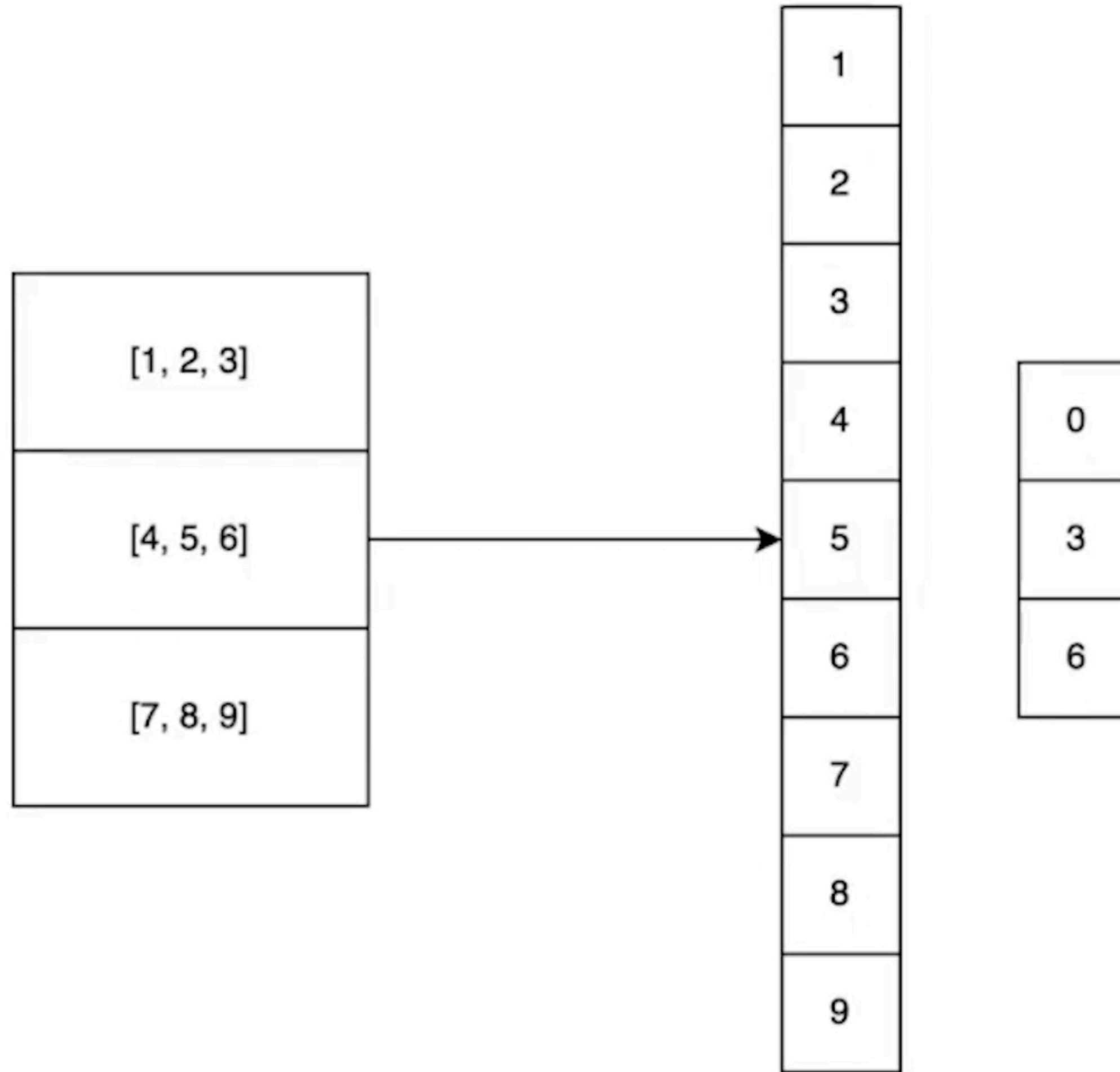
# Apache Parquet



# Сложные типы



# Сложные типы





Колоночные форматы - это  
хорошо для Big Data

# One more thing - Partition Discovery

```
path
└ to
  └ table
    └ gender=male
      └ ...
      └ country=US
        └ data.parquet
      └ country=CN
        └ data.parquet
      └ ...
    └ gender=female
      └ ...
      └ country=US
        └ data.parquet
      └ country=CN
        └ data.parquet
      └ ...
```

```
root
|--- name: string (nullable = true)
|--- age: long (nullable = true)
|--- gender: string (nullable = true)
|--- country: string (nullable = true)
```

# **4 - Mass Parallel Processing**

# Мотивация

- › Мы все еще учимся эффективно хранить и обрабатывать большие данные
- › Мы пришли к HDFS/MapReduce, но это медленно и не соответствует ACID от реляционных СУБД
- › А что если распараллелить вычисления, но сделать так, чтобы данные обрабатывали хосты, похожие на классические СУБД

# Мотивация

- › Мы все еще учимся эффективно хранить и обрабатывать большие данные
- › Мы пришли к HDFS/MapReduce, но это медленно и не соответствует ACID от реляционных СУБД
- › А что если распараллелить вычисления, но сделать так, чтобы данные обрабатывали хосты, похожие на классические СУБД
- › Так вот оказывается, что параллельно с DFS/MR разрабатывался класс систем, которые именно это и делают - MPP

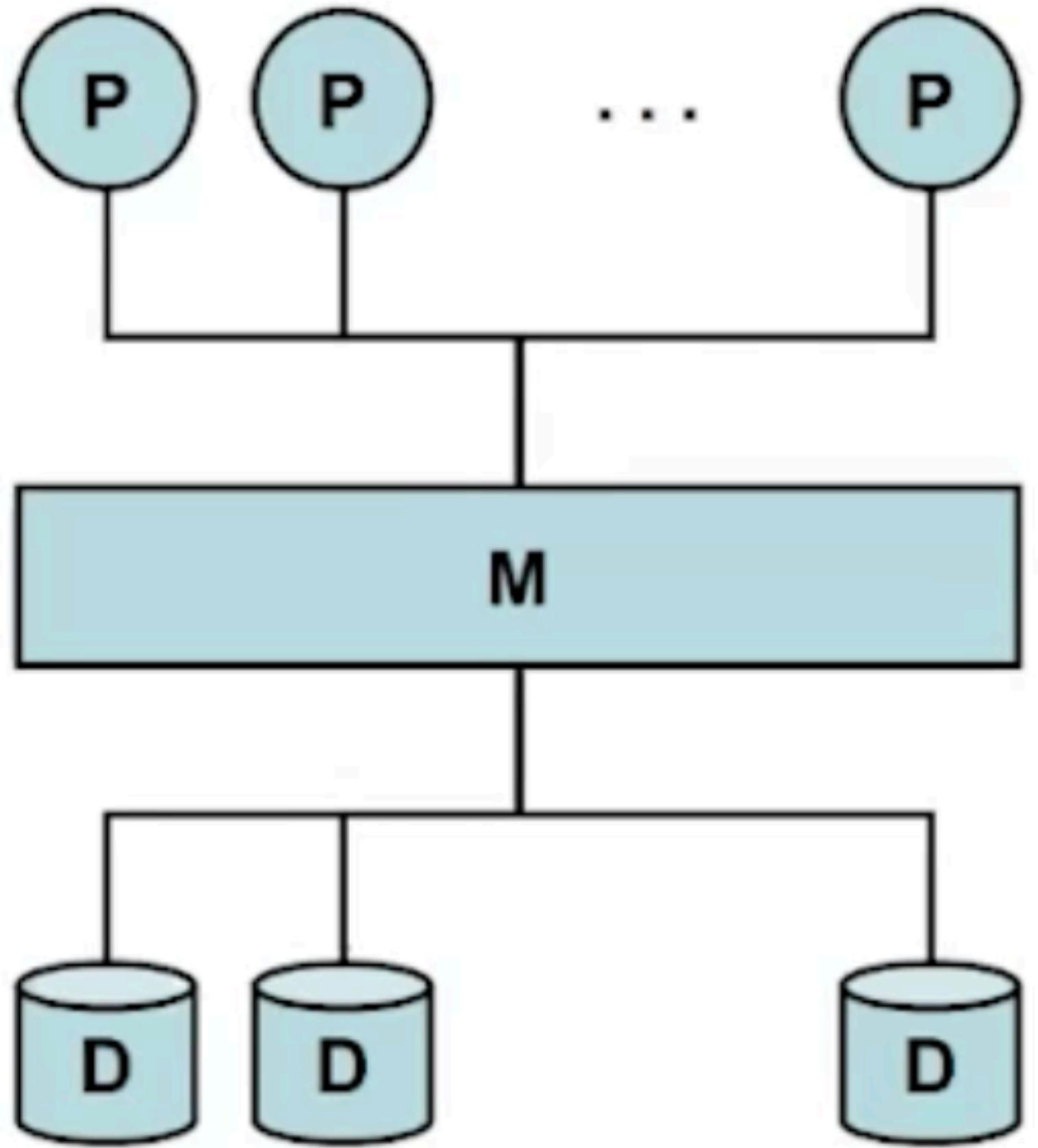
# Классификация архитектур

Мы хотим создать распределенную СУБД для эффективного хранения и обработки данных.

Какие тут могут быть подходы:

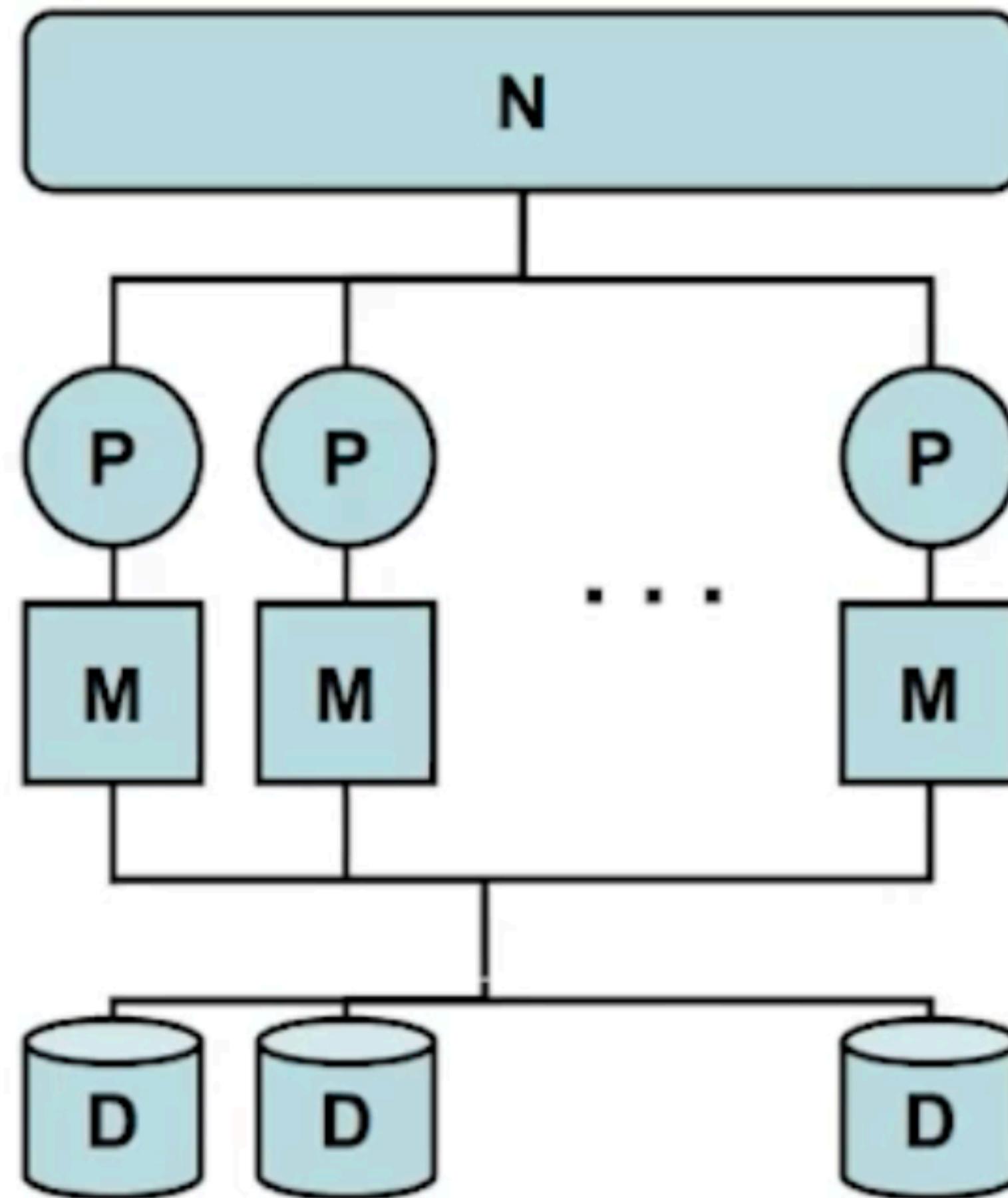
- › SE - Shared Everything - хосты делят все ресурсы (CPU, RAM, Disk)
- › SD - Shared Disks - хосты делят только диски
- › SN - Shared Nothing - нет совместного использования ресурсов
- › CD - Clustered Disk
- › CE - Clustered Everything

# SE (Shared Everything)



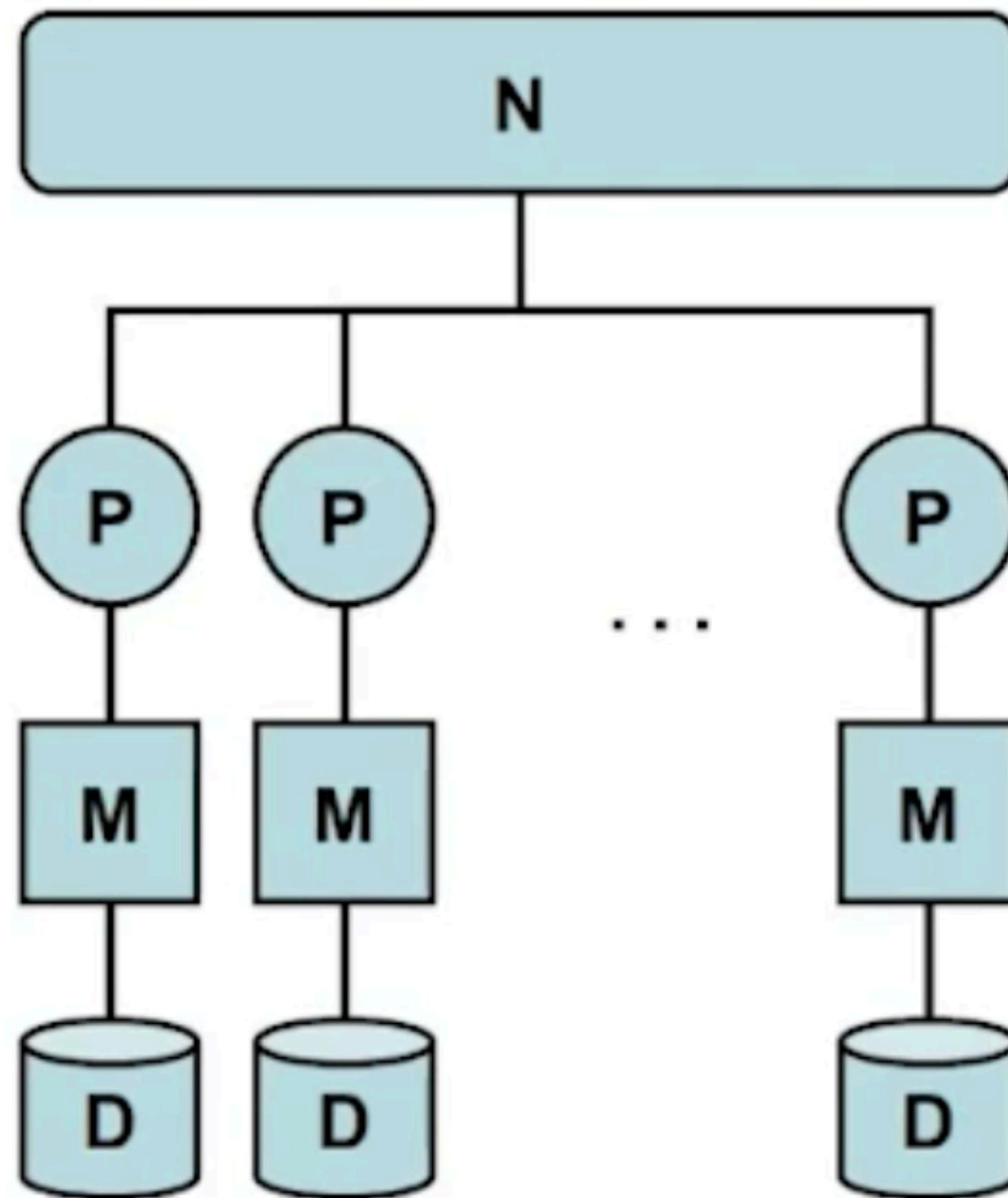
- › У всех процессоров общая RAM
- › Все диски доступны в равной степени и приоритете всем процессорам
- › Межпроцессорные коммуникации - через RAM
- › По сути - наш обычный компьютер

# SD (Shared Disk)



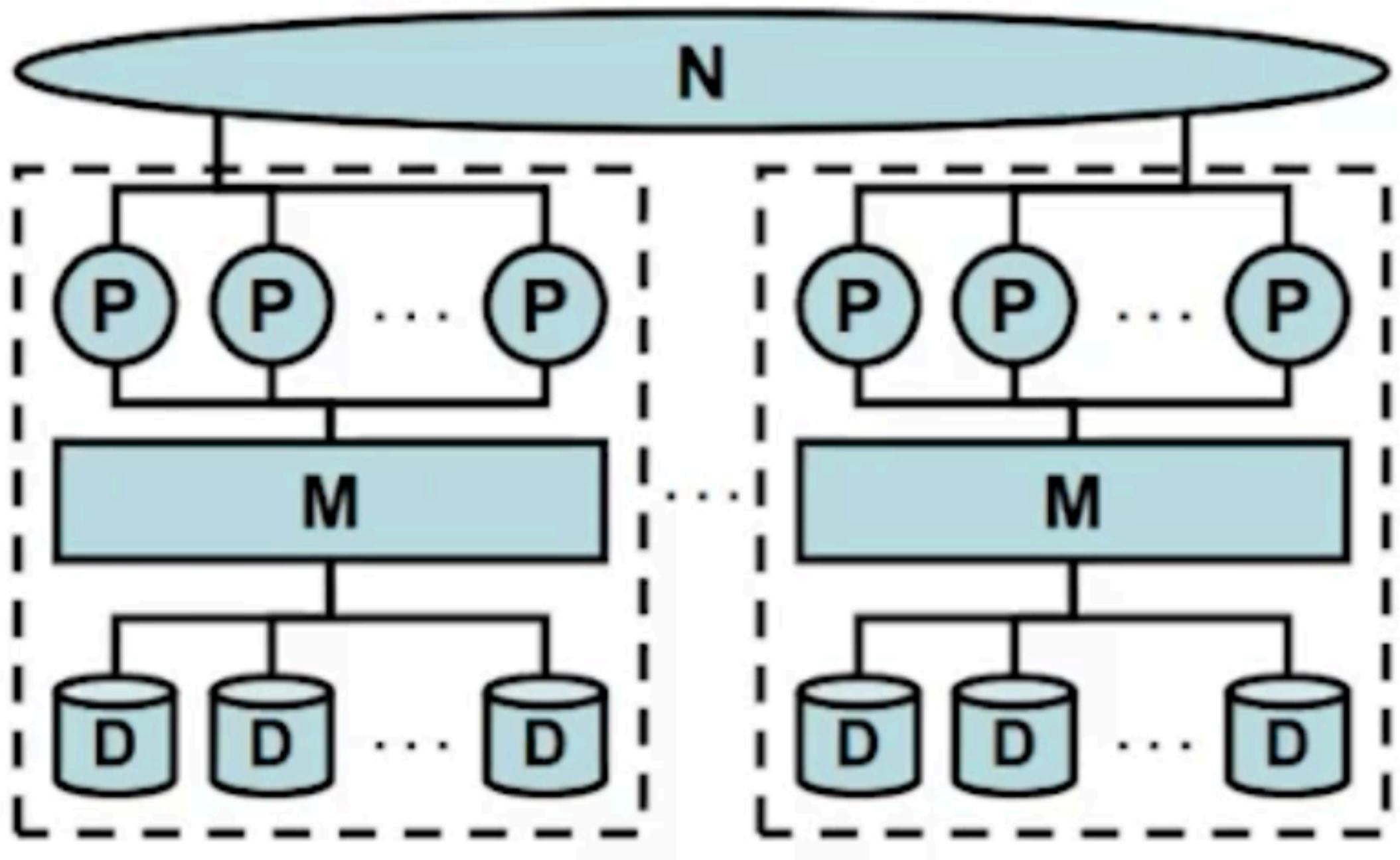
- › У каждого процессора своя RAM
- › Все диски доступны в равной степени и приоритете всем процессорам
- › Межпроцессорные коммуникации - через IO

# SN (Shared Nothing)



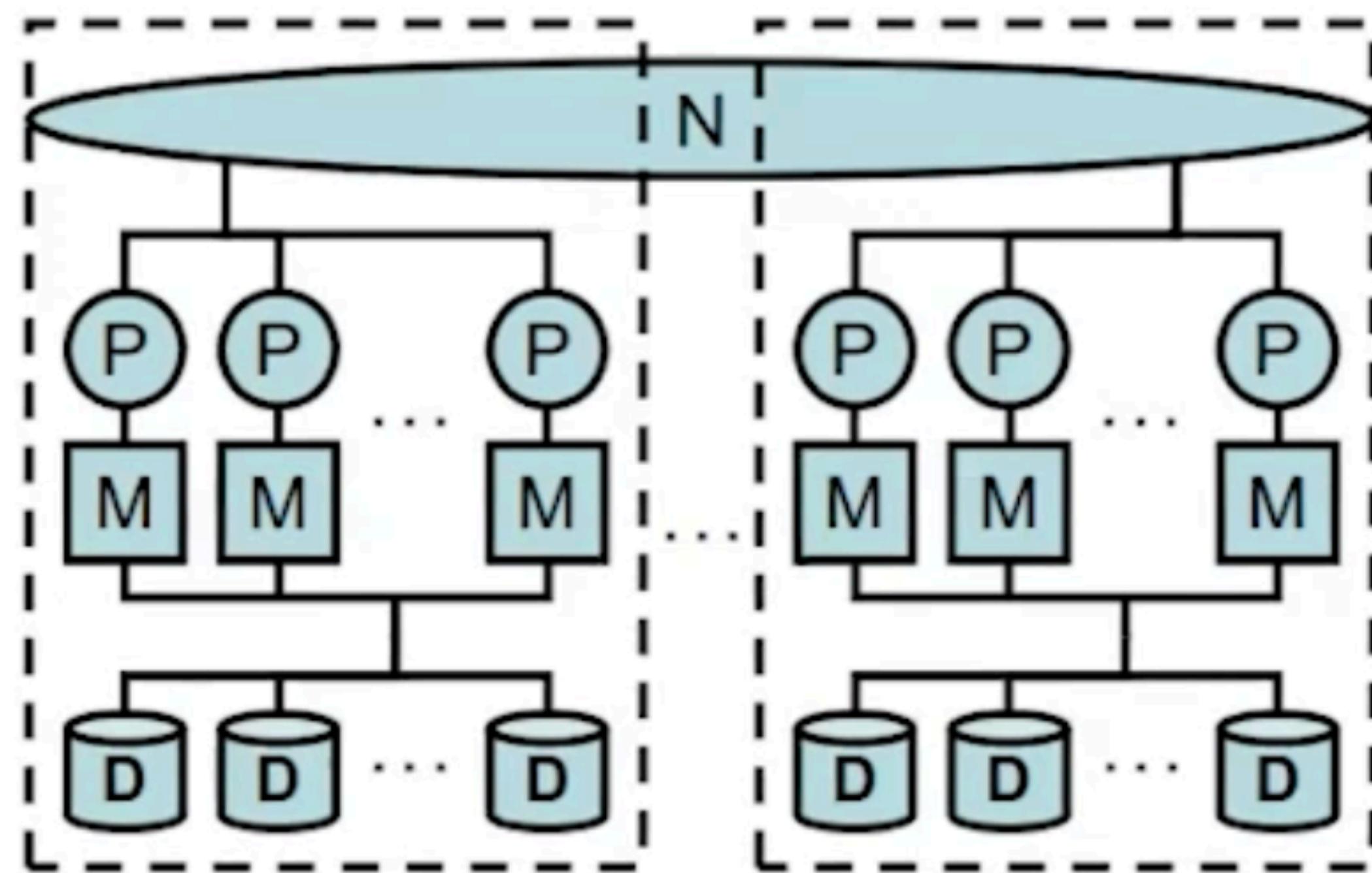
- › Каждый процессор имеет свою оперативную память и диск
- › Межпроцессорные коммуникации - через IO
- › Базовая архитектура всех MPP-расчетов: Hadoop, Vertica, GreenPlum и так далее

# CE (Clustered Everything)



- › Несколько SE-кластеров, связанных сетью
- › Межпроцессорные коммуникации - через RAM
- › МежклUSTERНЫЕ коммуникации - через IO

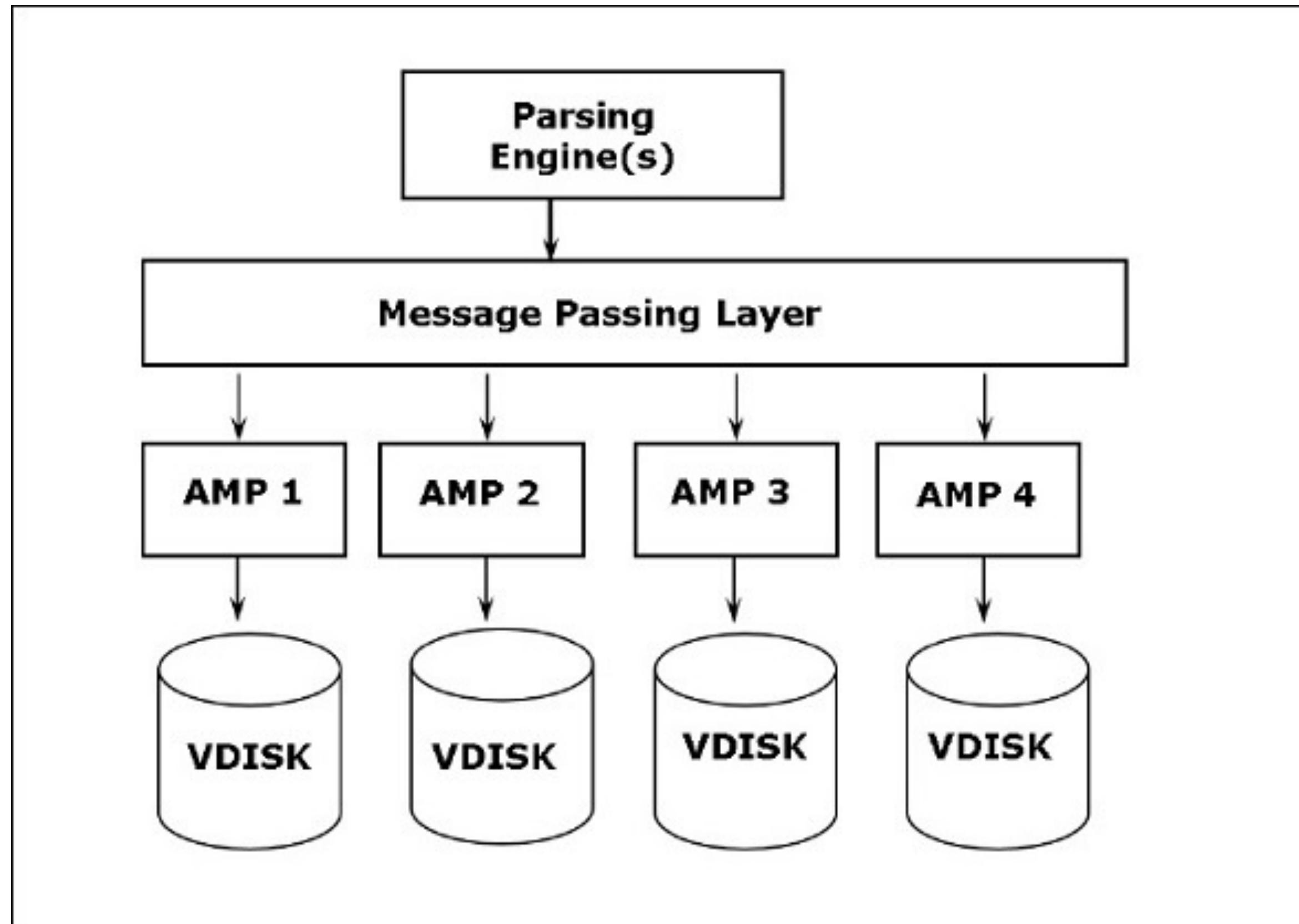
# CD (Clustered Disk)



- › Несколько SD-кластеров, связанных сетью
- › Межпроцессорные коммуникации - через локальную сеть
- › Межклusterные коммуникации - через глобальную сеть

# **5 - Архитектуры МРР**

# Terradata



## Parsing Engines:

- › Управление сессиями пользователей
- › Чтение и оптимизация запросов
- › Выдача результатов пользователю

## Message Processing Layer:

- › Посредник между РЕ и AMP

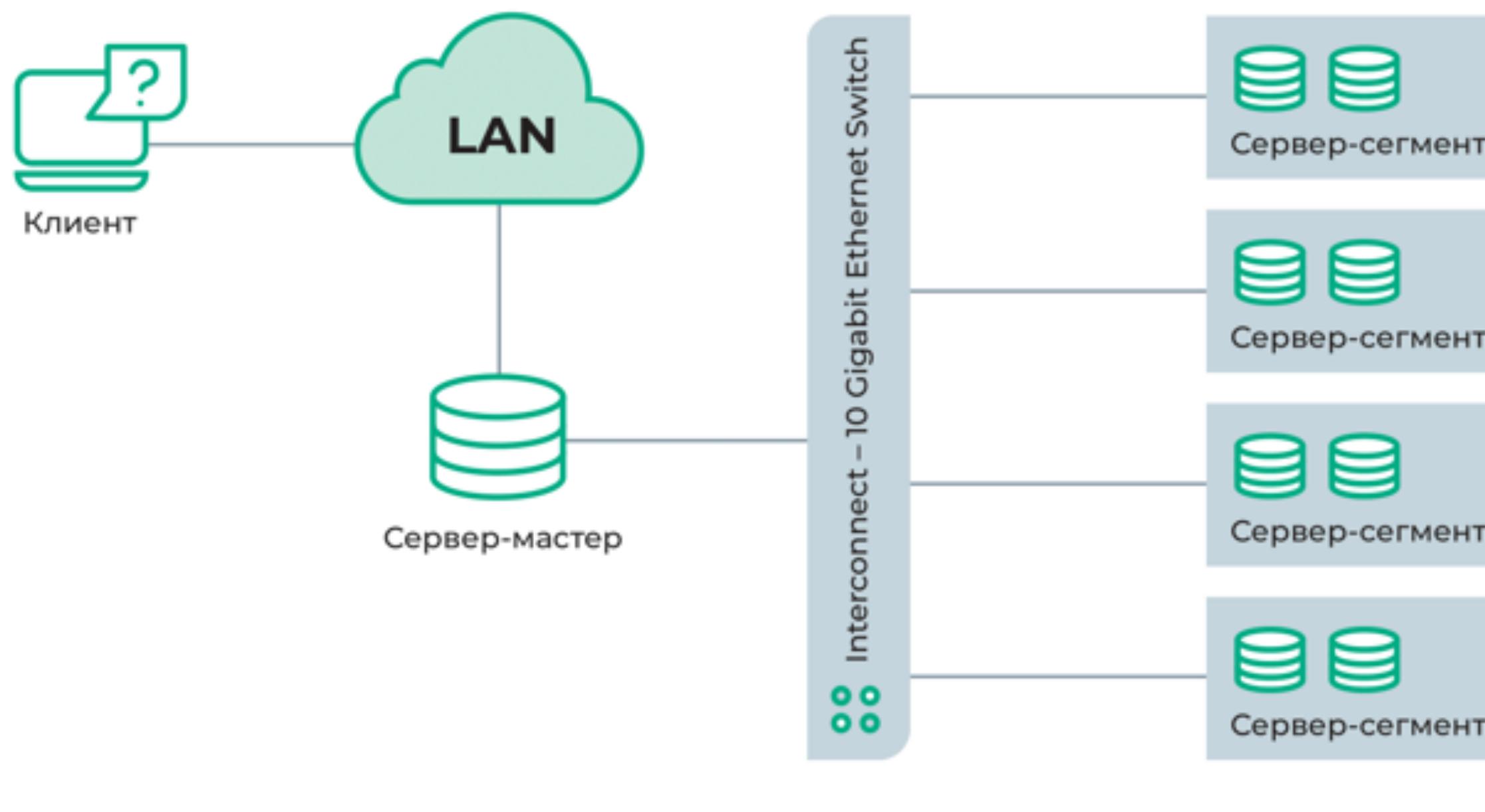
## Access Module Processor:

- › Управляет выделенным пространством
- › Выполняет отправленные операции

## Virtual Disks:

- › Непосредственно выделенное дисковое пространство

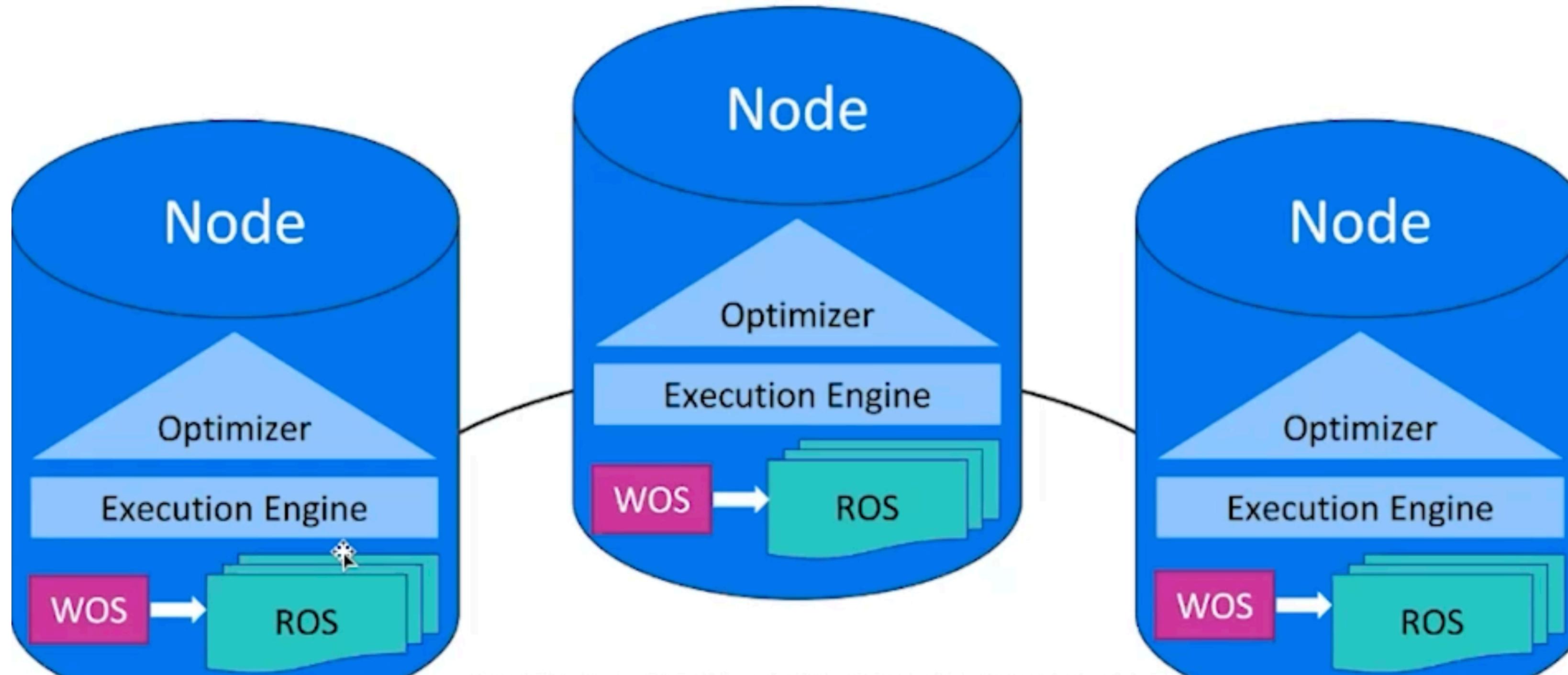
# Greenplum



- › В GP все чуть хуже - есть одна (или две) мастер-ноды - это точка отказа и бутылочное горлышко в системе
- › Внутри сегментов по сути лежат маленькие постгрессы
- › Умеет как в локальные, так и в распределенные JOIN
- › Полностью соответствует ACID
- › Нативно поддерживает постгресовые приколы (PostGIS, нативная репликация по CDC)
- › Полностью соответствует ACID
- › Умеет в масштабирование “на горячую”

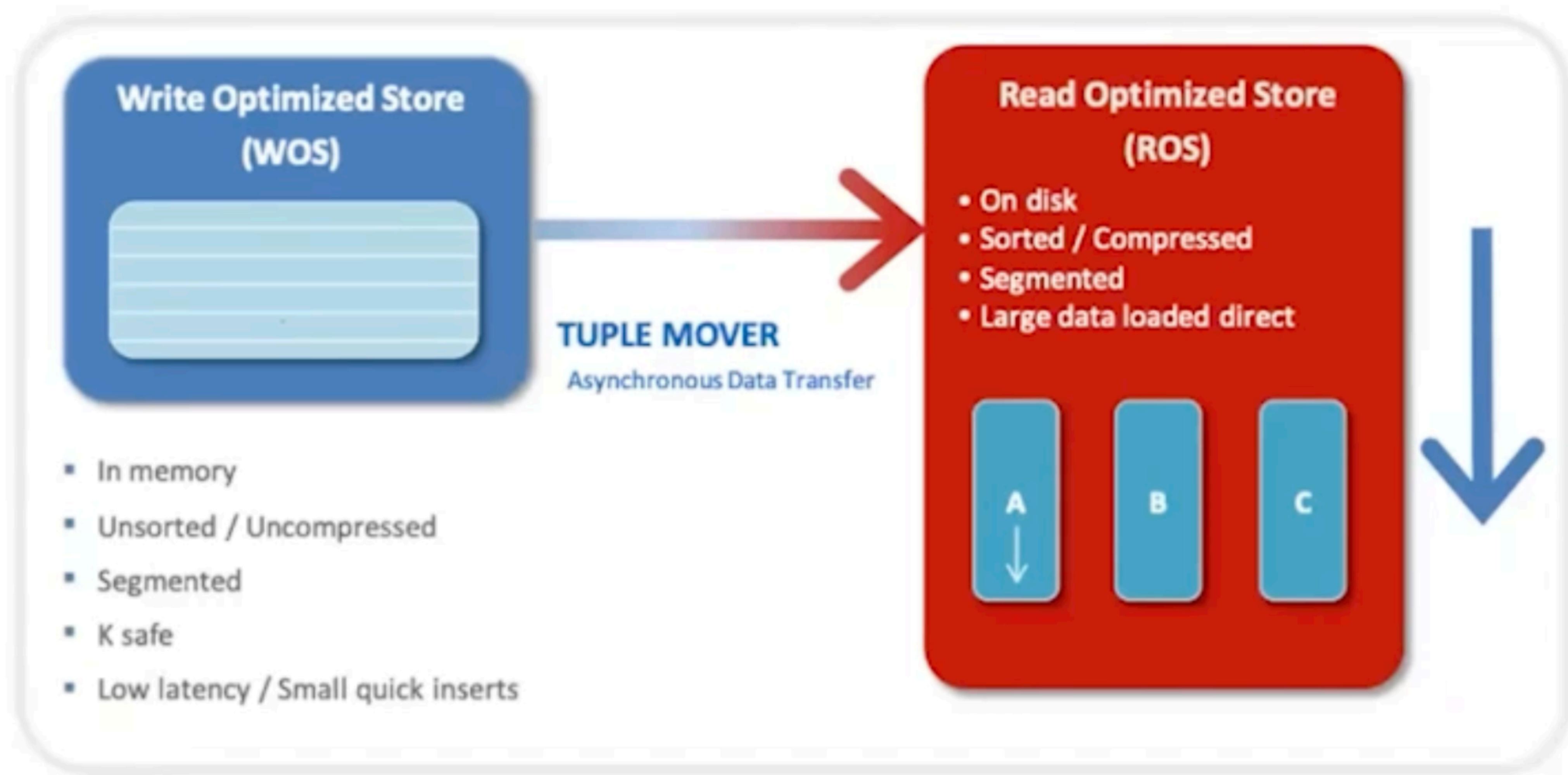
Pic. 3

# Vertica



- › Похожа на Terradata
- › Исключительно колоночная СУБД (!)
- › Очень гибко администрируется
- › Шикарный инструмент проекций
- › WOS / ROS - память

# Vertica



- In memory
- Unsorted / Uncompressed
- Segmented
- K safe
- Low latency / Small quick inserts

# Фрагментарный параллелизм

Во всех перечисленных системах есть способ задать распределение по узлам:

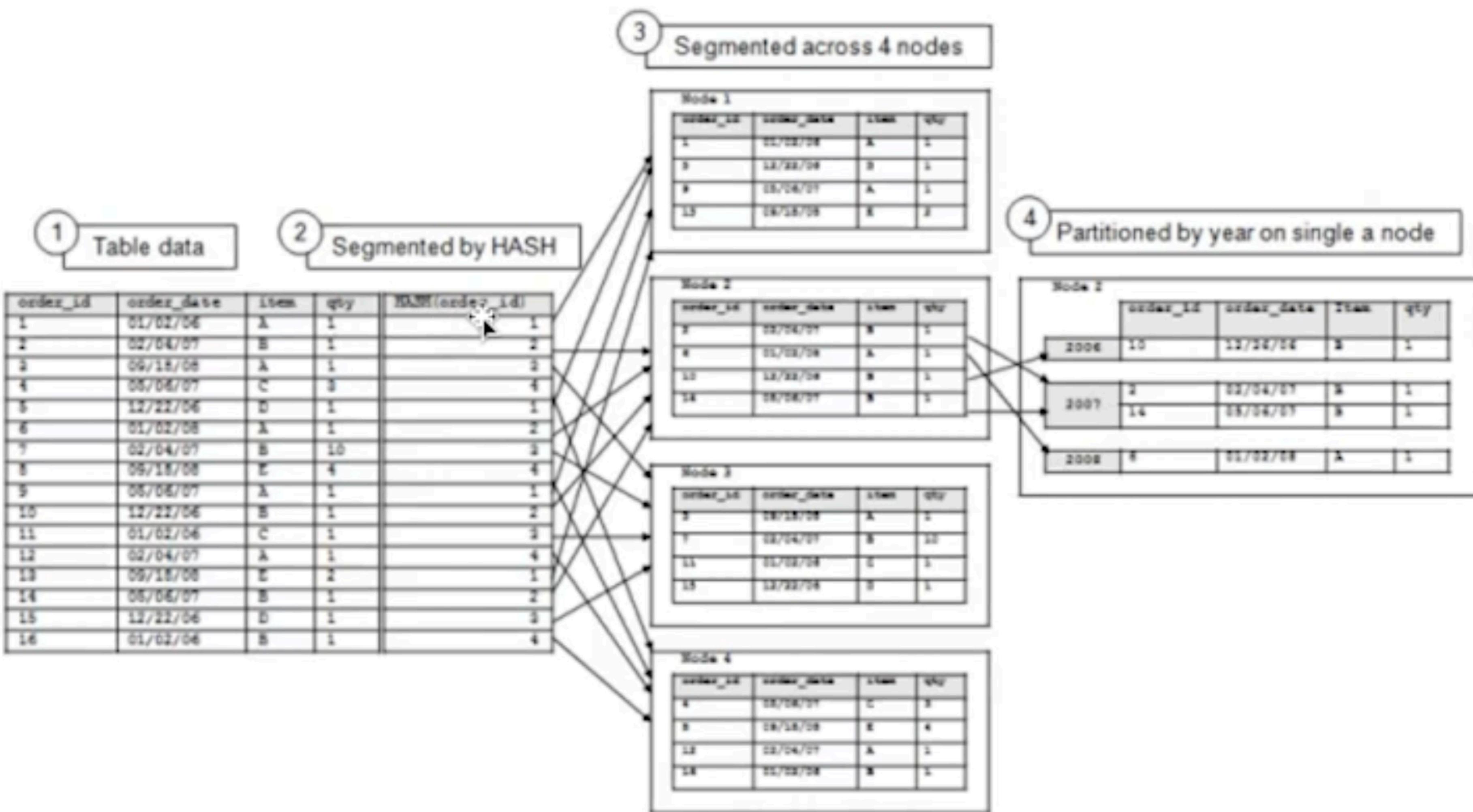
- › Terradata - primary index
- › Greenplum - distribution key
- › Vertica - segmentation key

# Партицирование

Физическое разделение данных на диске по какому-то полю

- › Данные разделяются по нодам в зависимости от ключа сегментации
- › В каждой номе данные делятся на фрагменты по ключуパーティрования
- › Внутри партиции строки сортируются в зависимости от выбранного ключа сортировки
- › Ключパーティрования может не совпадать с ключом сегментации

# Сегментирование иパーティционирование вместе



# Проблема выбора ключа сегментации

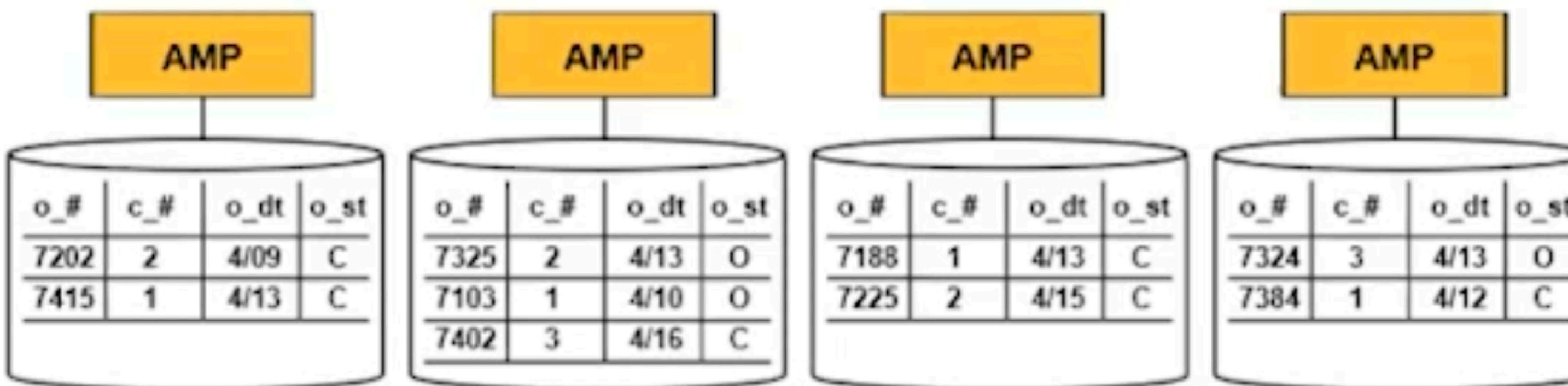
Критерии выбора ключа сегментации:

- › Доступ к данным ключа сегментации должен осуществляться чаще остальных
- › Распределение значений должно быть равномерным
- › В больших таблицах количество уникальных значений ключа должно превышать количество нод как минимум в 10 раз
- › Изменения в данных ключа сегментации не должны происходить слишком часто

# Пример

Orders

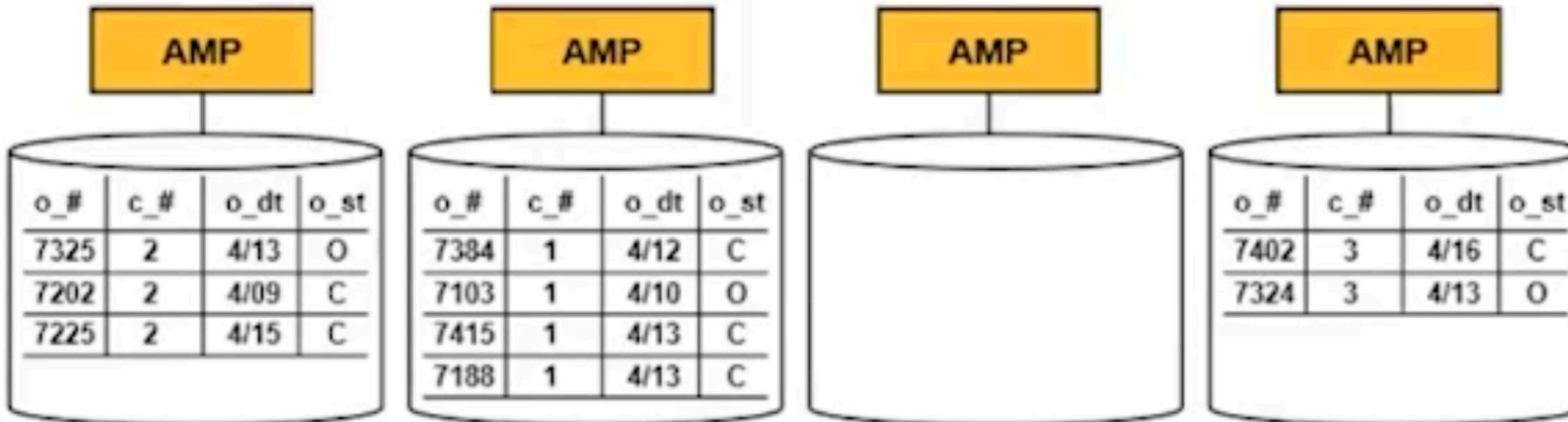
Order Number	Customer Number	Order Date	Order Status
PK			
UPI			
7325	2	4/13	O
7324	3	4/13	O
7415	1	4/13	C
7103	1	4/10	O
7225	2	4/15	C
7384	1	4/12	C
7402	3	4/16	C
7188	1	4/13	C
7202	2	4/09	C



# Пример

Orders

Order Number	Customer Number	Order Date	Order Status
PK			
	NUPI		
7325	2	4/13	O
7324	3	4/13	O
7415	1	4/13	C
7103	1	4/10	O
7225	2	4/15	C
7384	1	4/12	C
7402	3	4/16	C
7188	1	4/13	C
7202	2	4/09	C



# Пример

Orders

Order Number	Customer Number	Order Date	Order Status
PK			
			NUPI
7325	2	4/13	O
7324	3	4/13	O
7415	1	4/13	C
7103	1	4/10	O
7225	2	4/15	C
7384	1	4/12	C
7402	3	4/16	C
7188	1	4/13	C
7202	2	4/09	C

AMP

AMP

AMP

AMP

o_#	c_#	o_dt	o_st
7402	3	4/16	C
7202	2	4/09	C
7225	2	4/15	C
7415	1	4/13	C
7188	1	4/13	C
7384	1	4/12	C

o_#	c_#	o_dt	o_st
7103	1	4/10	O
7324	3	4/13	O
7325	2	4/13	O

# Проблема использованияパーティционирования

Плюсы:

- › Если на большой таблице примененоパーティционирование, то читается только необходимый блок данных

Минусы:

- › Дополнительные накладные расходы на чтение/запись файлов
- › Запросы, которые не используют колонкиパーティционирования, работают хуже
- › Не работает MergeJoin без использования колонокパーティционирования

# Фрагментарный параллелизм

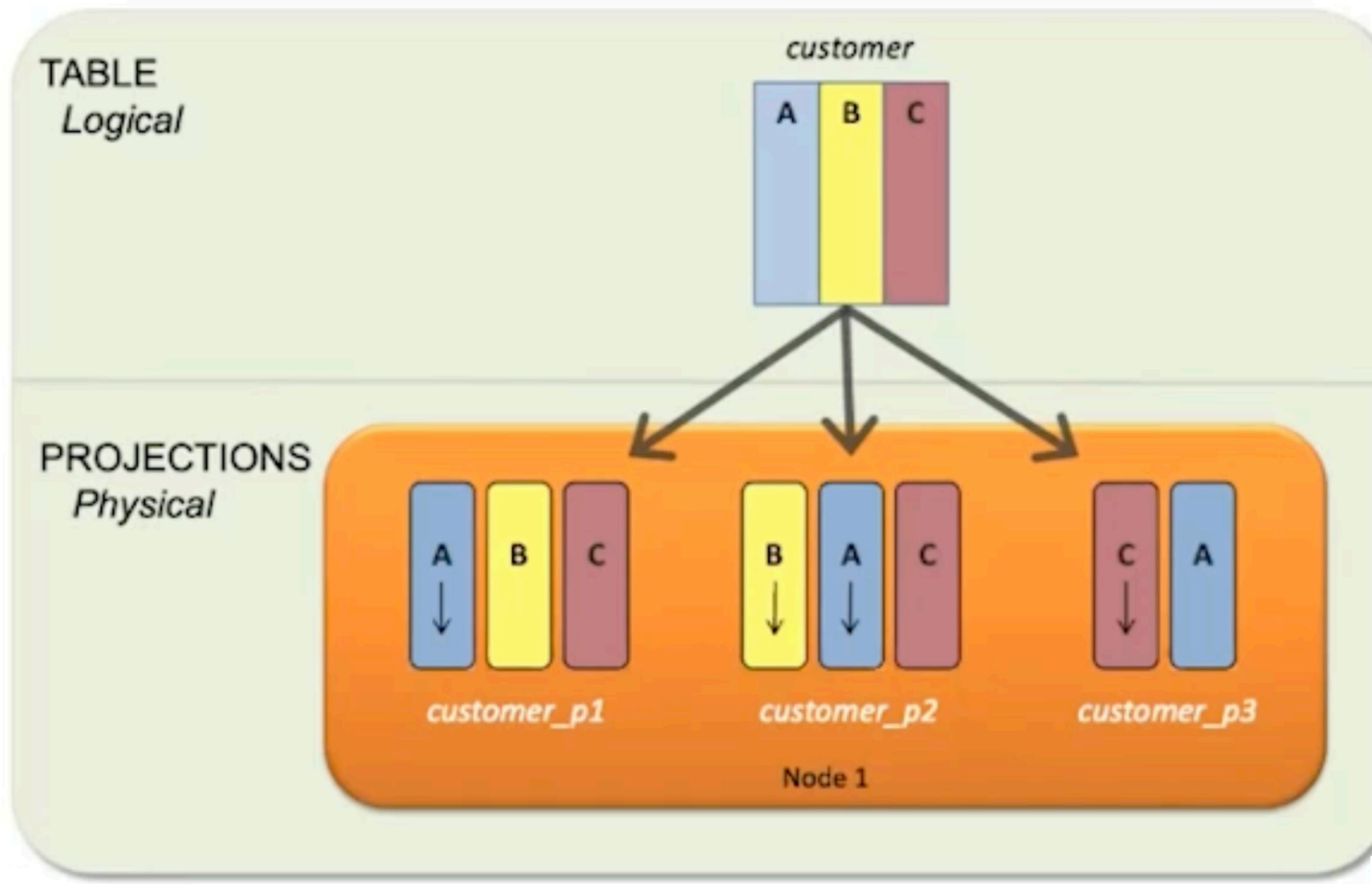
Во всех перечисленных системах есть способ задать распределение по узлам:

- › Terradata - primary index
- › Greenplum - distribution key
- › Vertica - segmentation key

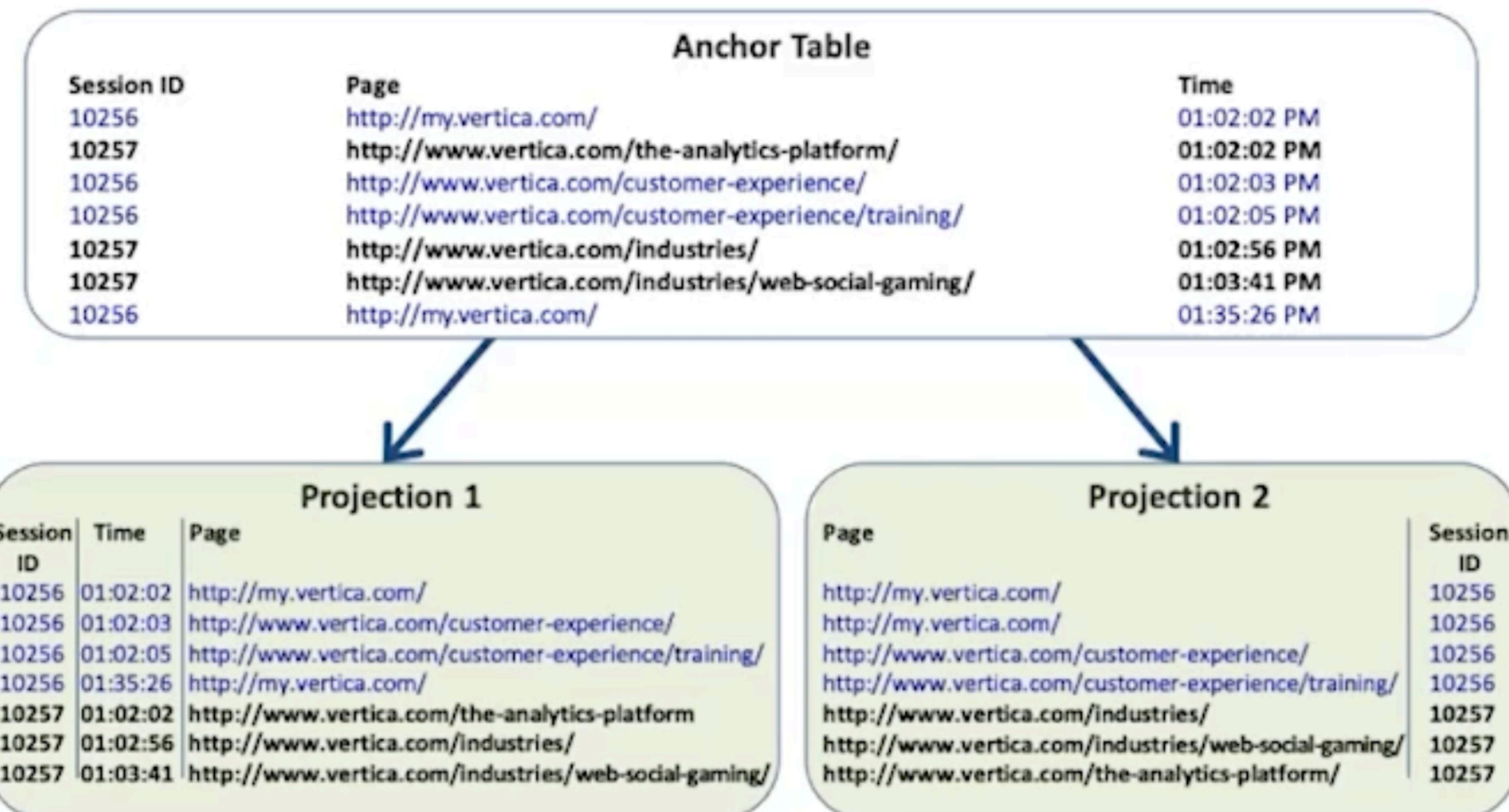
Методы доступа к данным:

- › Full scan - все МРР
- › Обращение по ключу сегментации - все МРР
- › Использование проекций - Vertica
- › Использование вторичного индекса - Greenplum

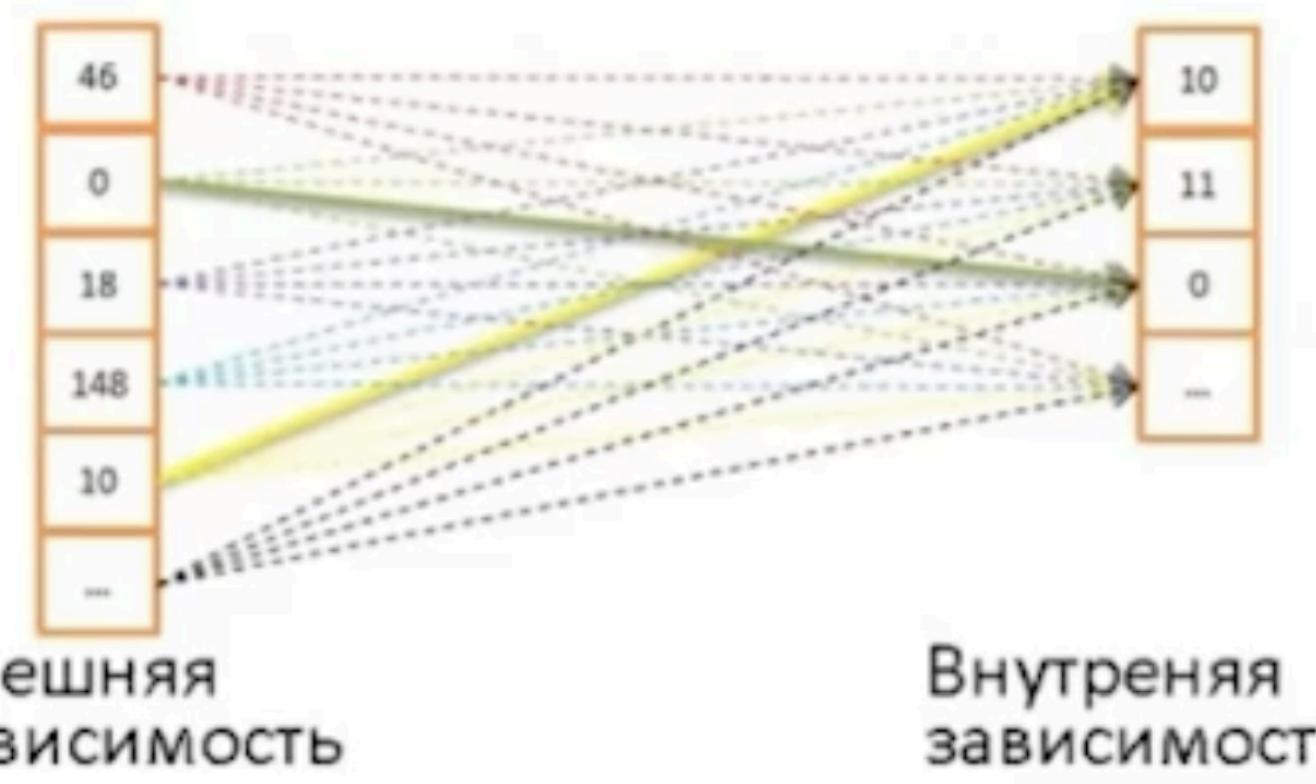
# Проекции



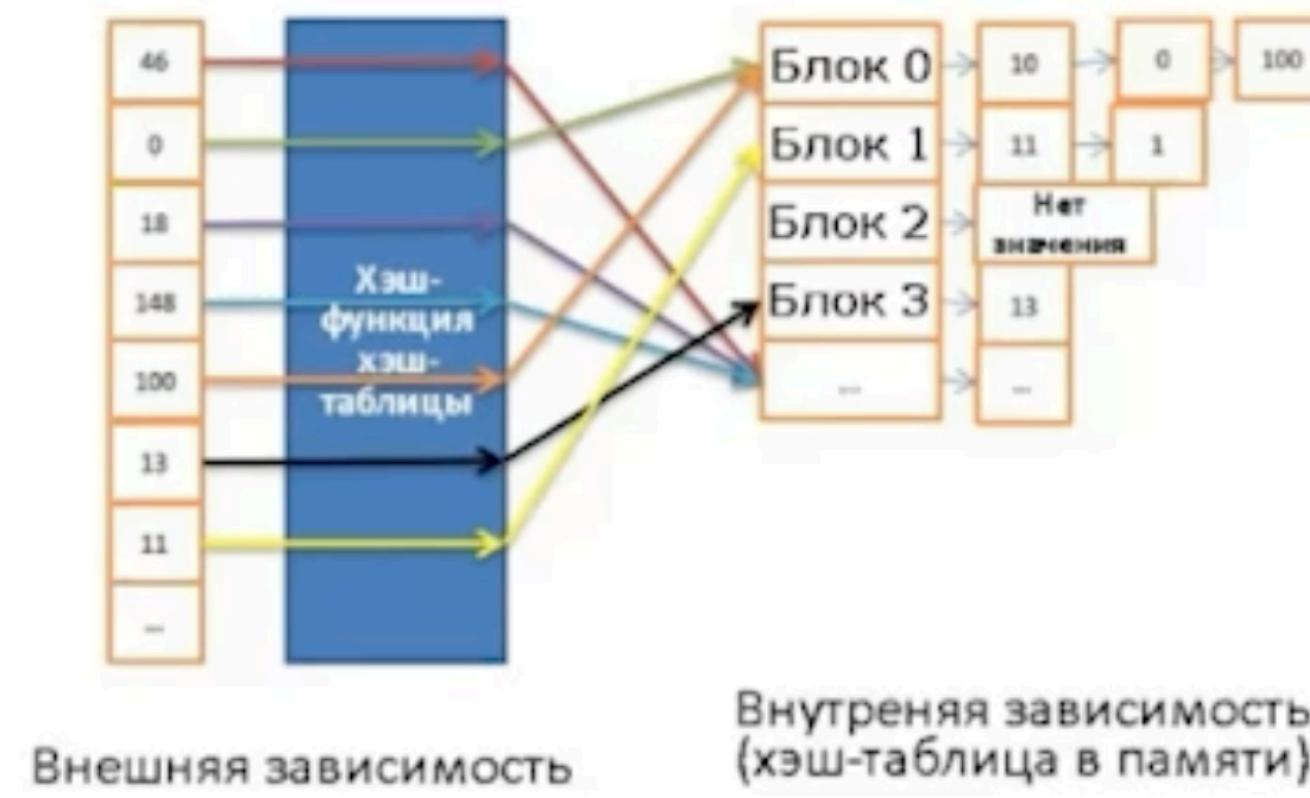
# Проекции



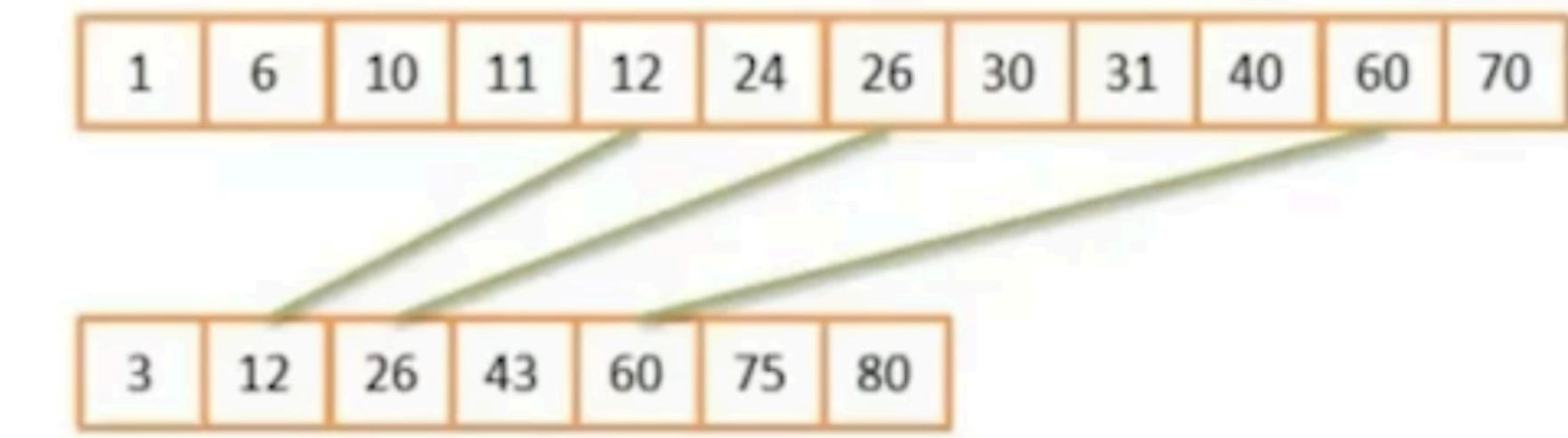
# Виды JOIN



**Nested loop**



**Hash join**



**Merge join**

# Merge join на МРР

- › Необходимо, чтобы данные были одинаково распределены
- › Блоки таблицы читаются только один раз
- › В большинстве случаев быстрее других join

Как работает на стороне БД:

- › Находим меньшее множество
- › Решаем, что с ним делать - перераспределение/дублирование/сортировка
- › Делаем join

# ИТОГ

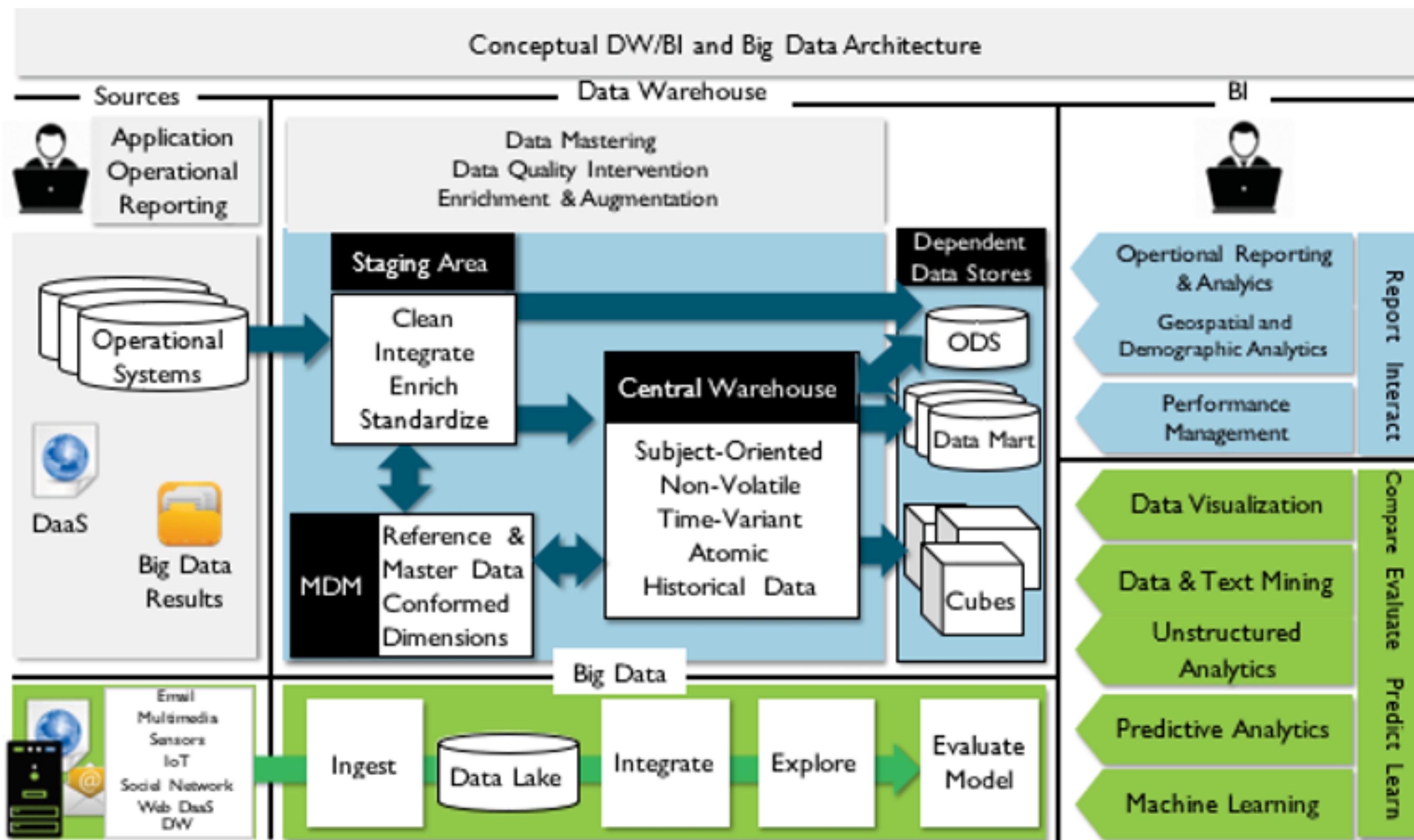


Figure 5: Date Warehouse Concept