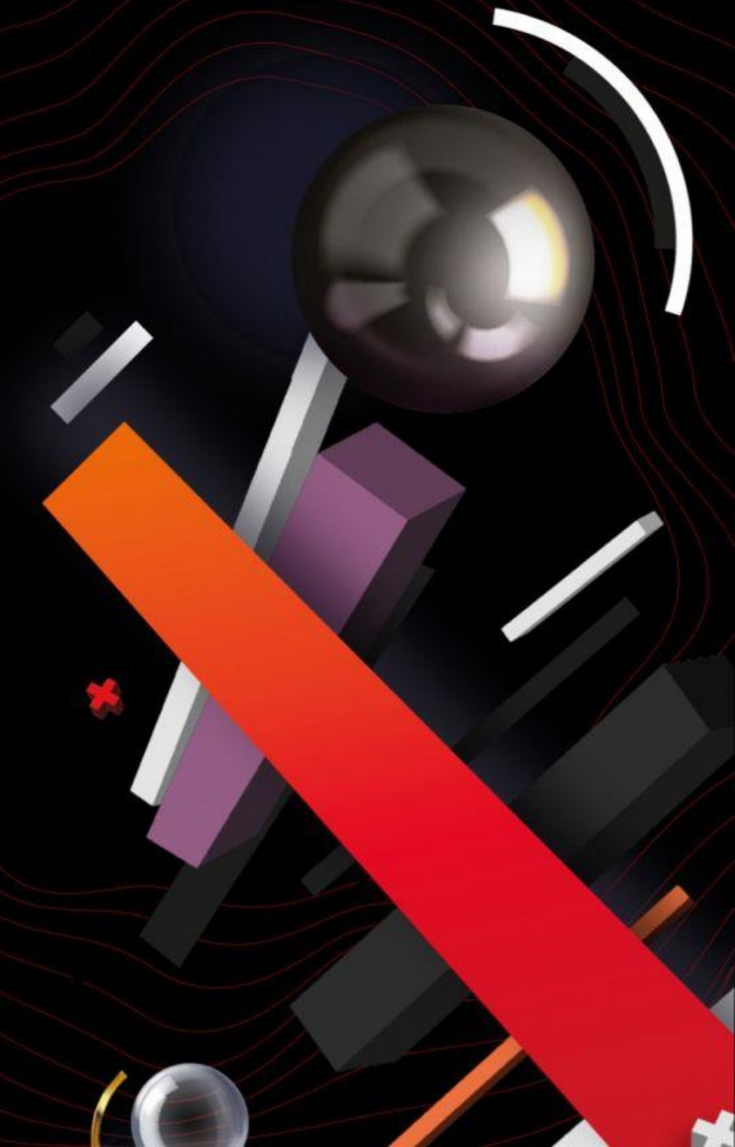
Реализовать OLAP

Как мы делали колоночное хранение в YDB

Новожилова Софья, Яндекс

HL++ 2023 HighLoad++



Что такое YDB и где используется

YDB – это СУБД, которая родилась
и выросла внутри Яндекса.

Распределенная

Масштабируемая

Производительная

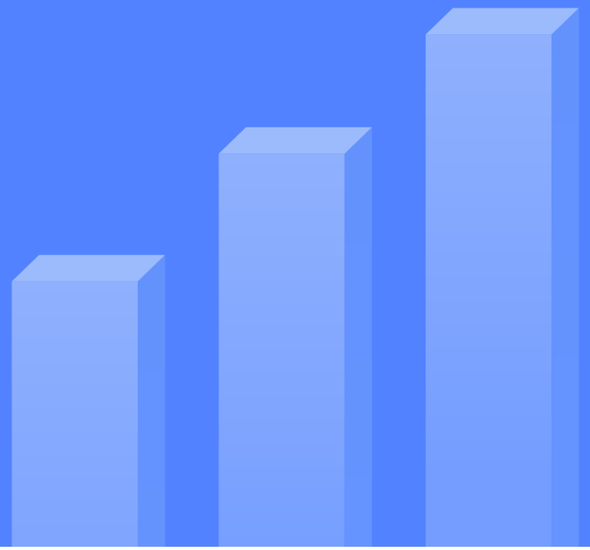
Yandex Cloud Банк

Маркет Пэй ID

Что умеет YDB

Транзакционные нагрузки

Аналитические нагрузки

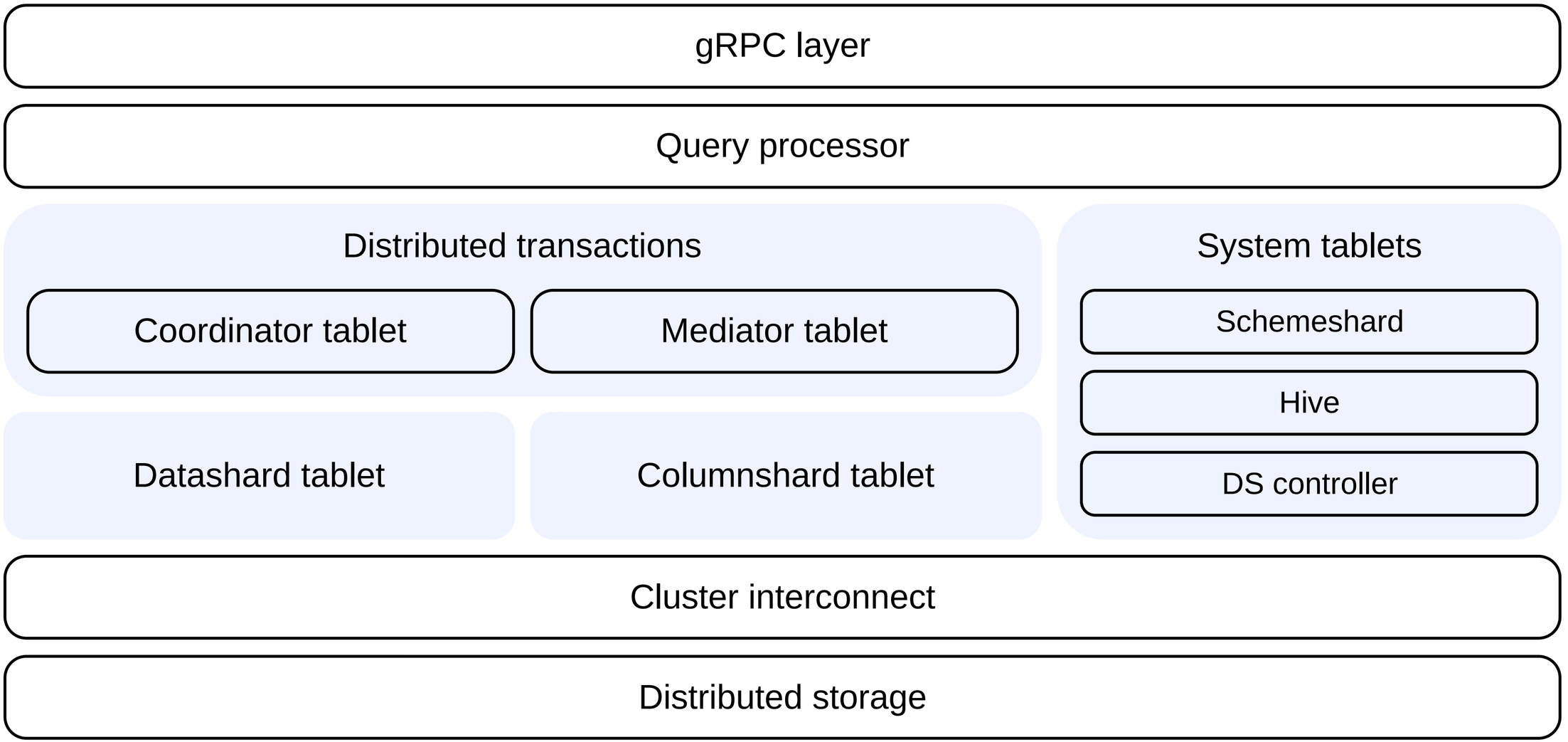


Стриминг данных

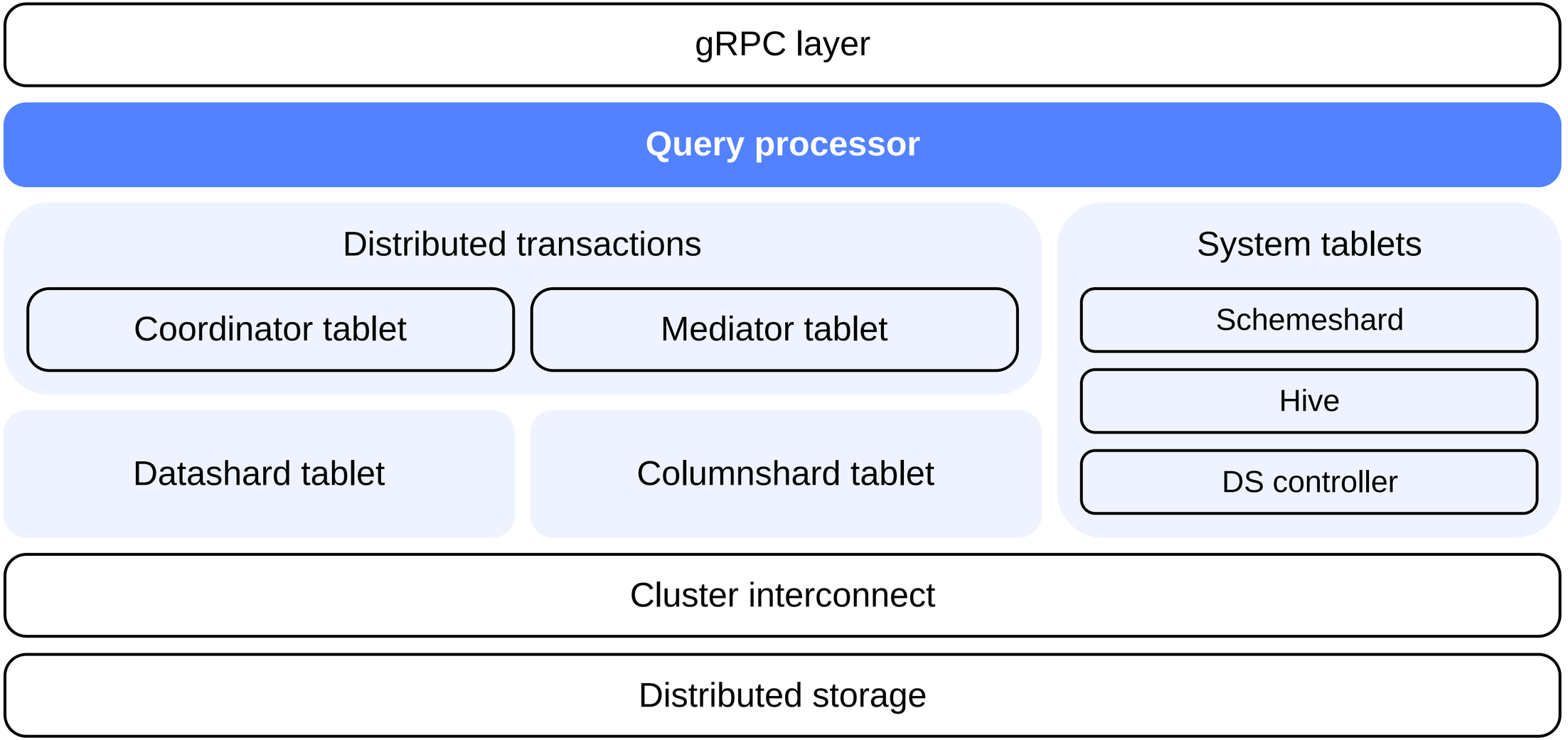
О чем будет доклад

- Поделюсь опытом встраивания нового движка в существующую платформу;
- Расскажу, какую модель мы выбрали для хранения данных;
- Какие задачи из этого возникли;
- Как мы их решали.

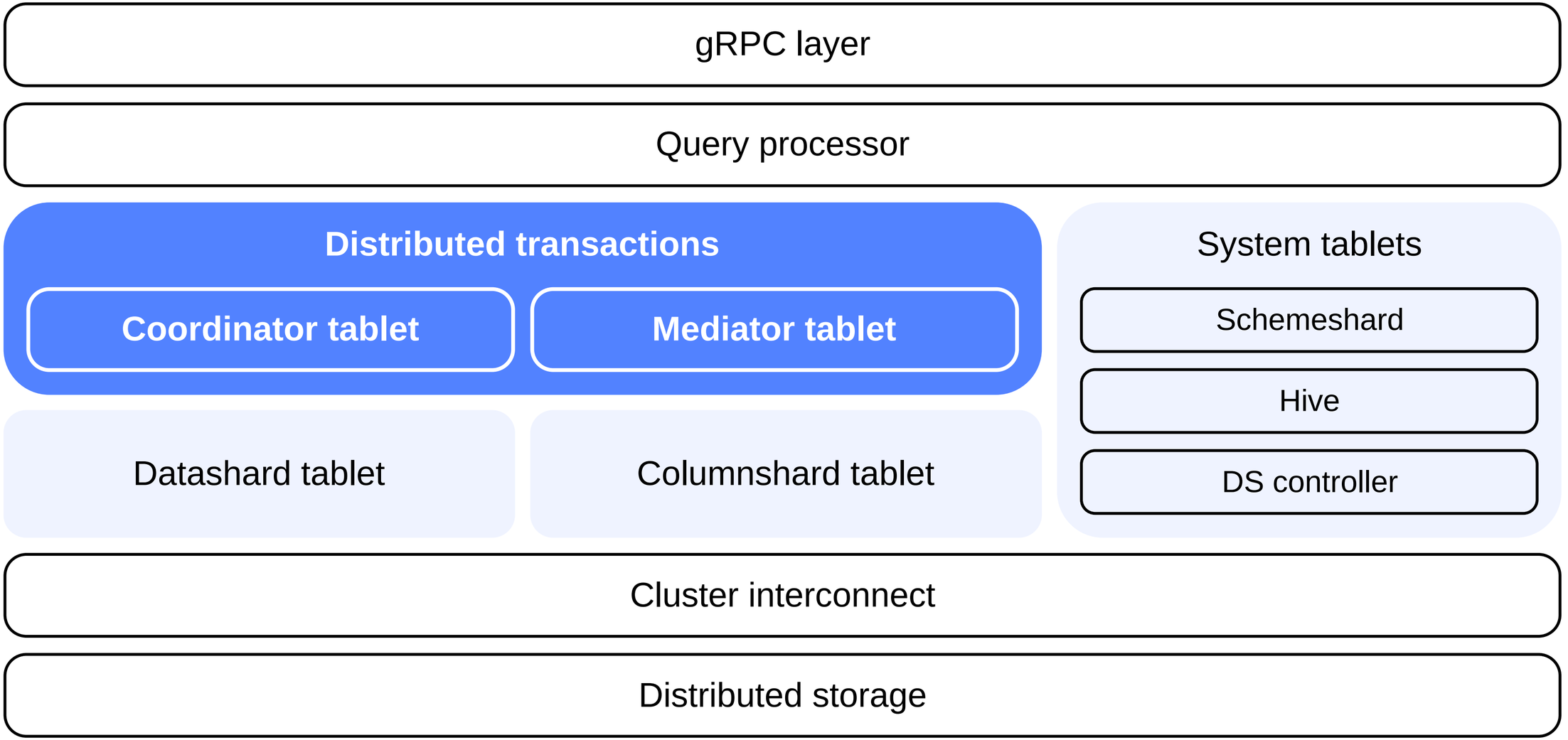
Архитектура за 3 минуты



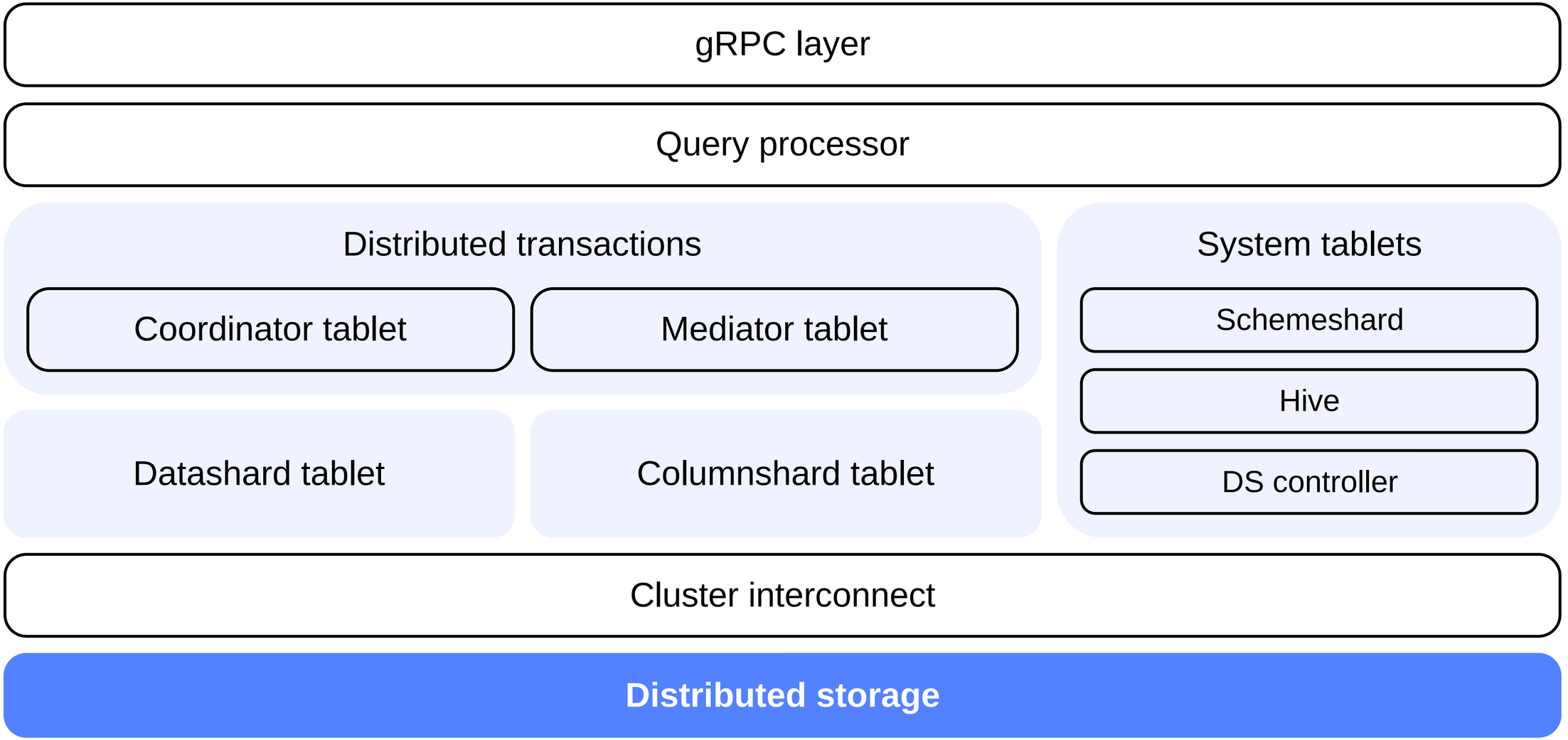
Процессинг запросов



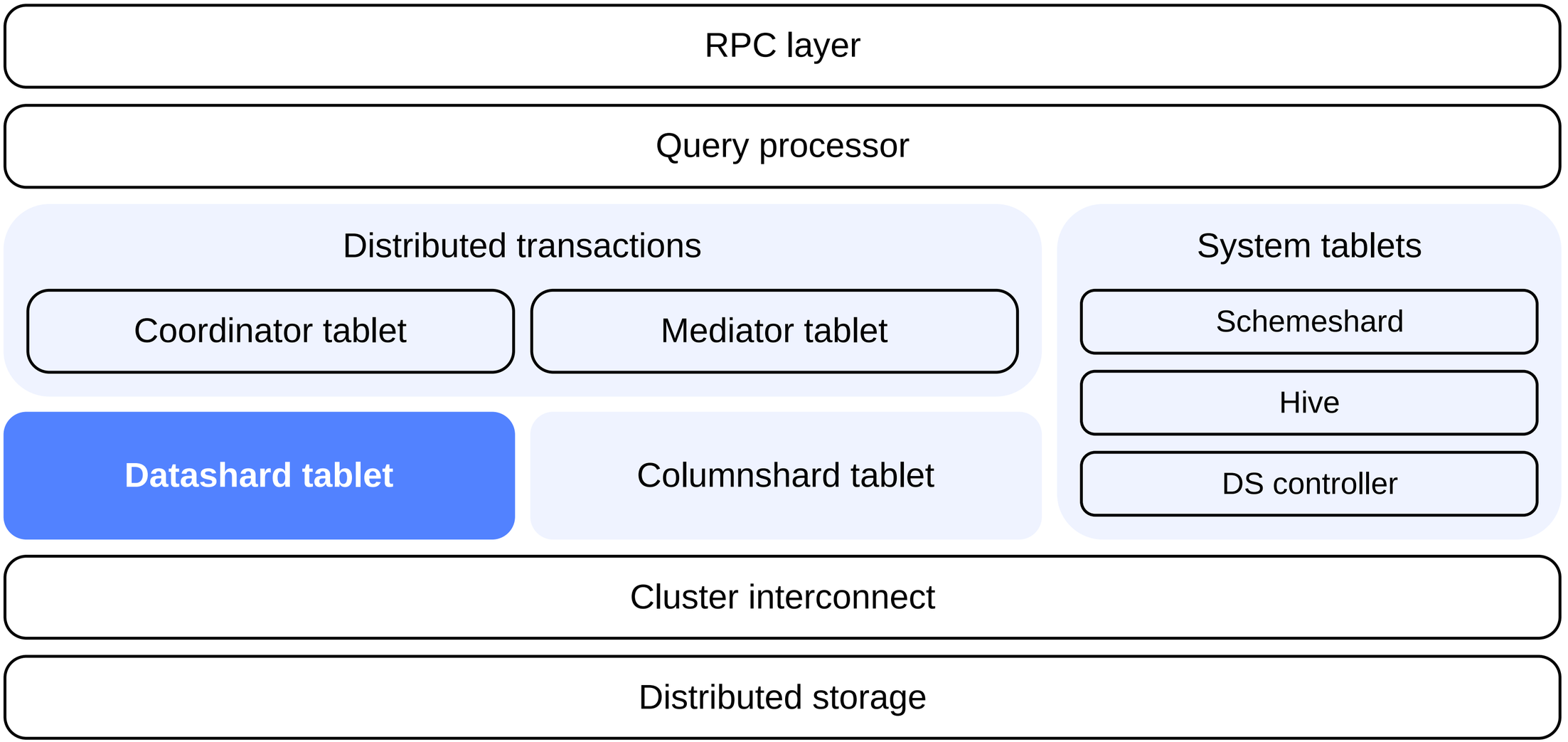
Подсистема транзакций



Распределенное хранилище



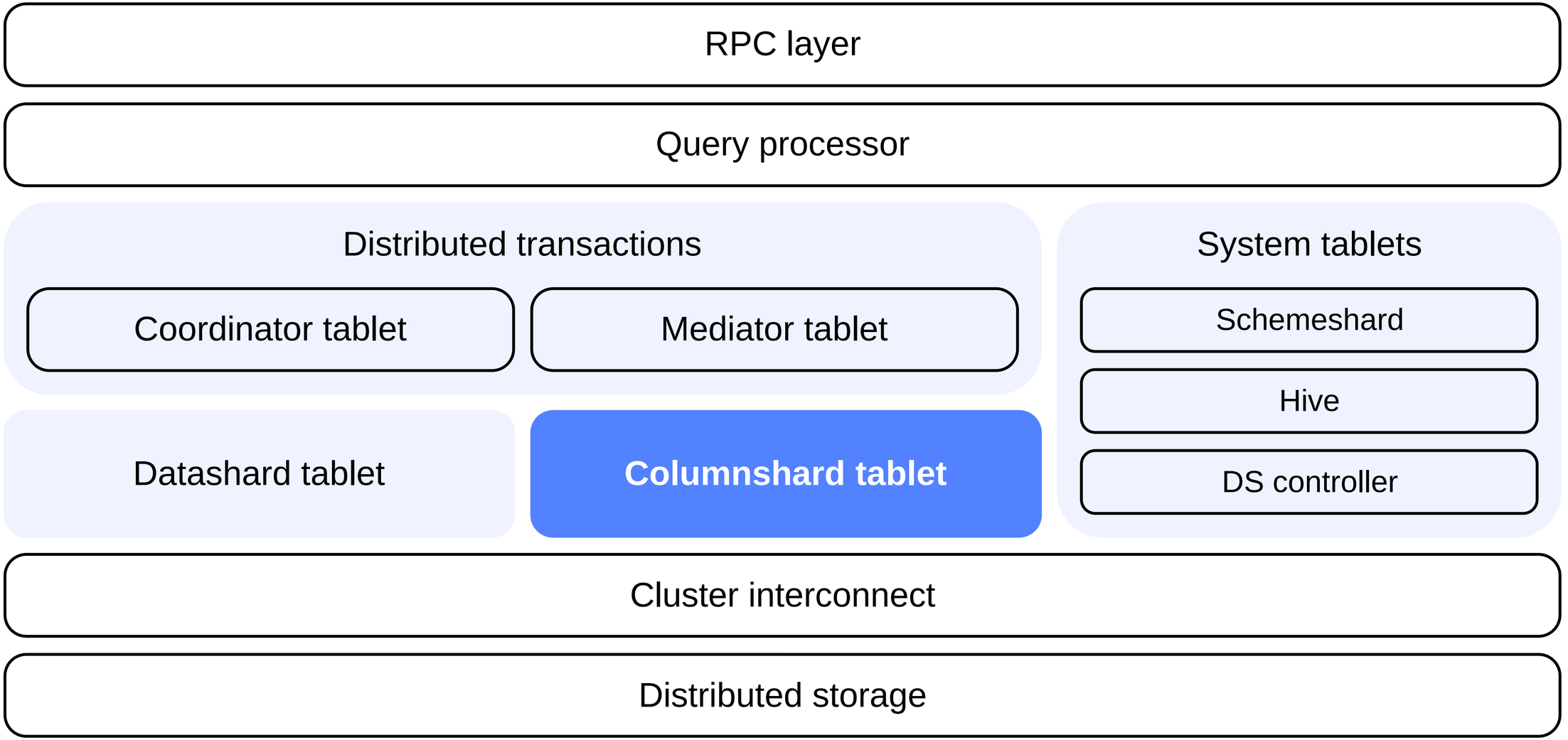
Метаданные таблицы



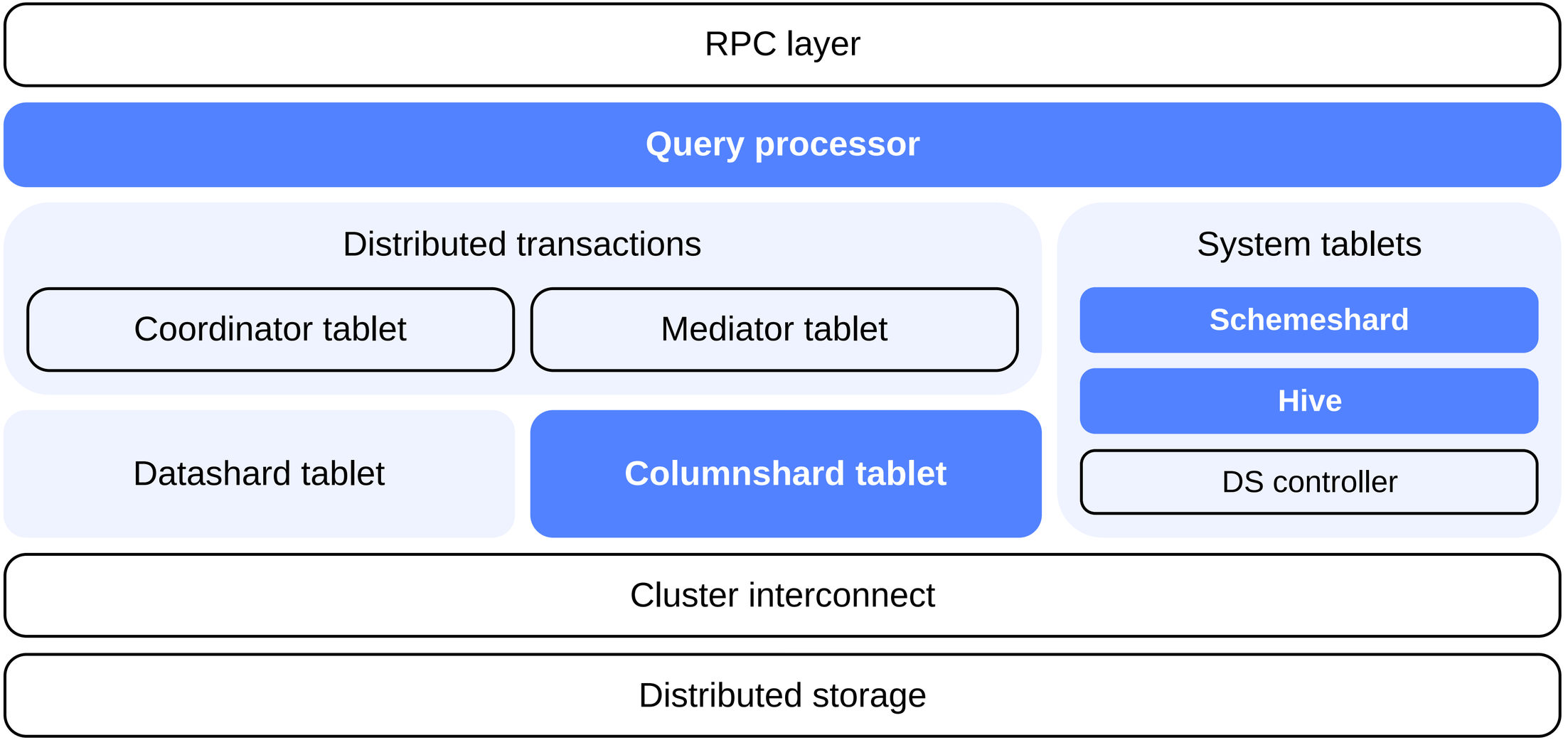
Добавление новой функциональности

Ожидания

Добавим метаданных



Подтюним общие компоненты

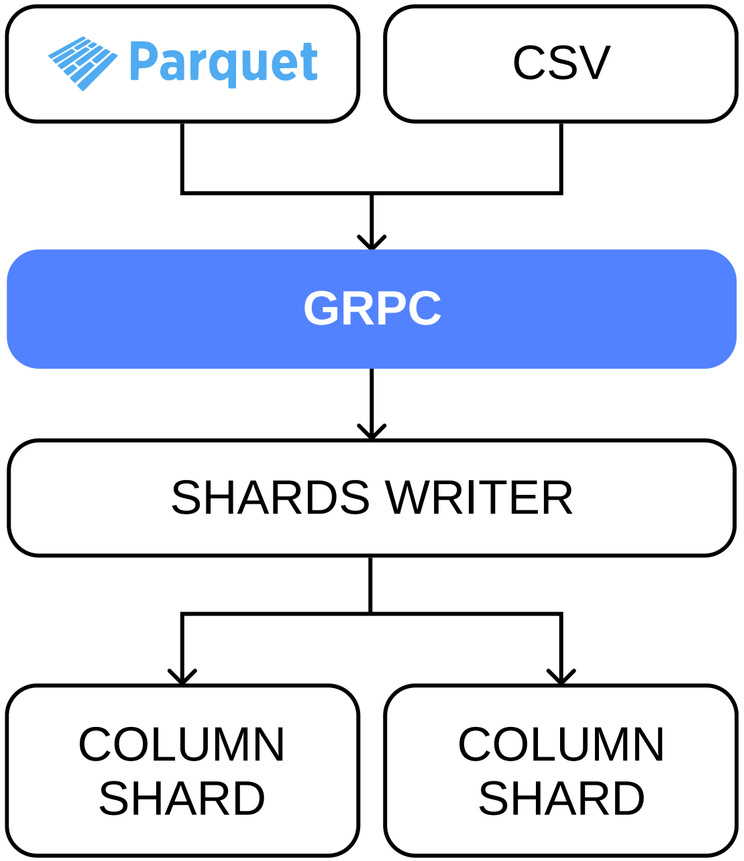


Какие задачи мы себе ставили

- Масштабирование как часть платформы;
- Минимум ручных настроек системы:
  - — Не нужно подбирать размеры входящих батчей;
  - — Не нужно задавать параметры внутренних процессов;
  - — Адаптироваться под профиль использования данных;
  - — Адаптироваться под характер хранимых данных.
- Сделать платформу для аналитики, с поддержкой транзакций между таблицами разных типов OLAP и OLTP;
- Жизнеспособное распределение ресурсов между OLAP и OLTP.

Заливка данных

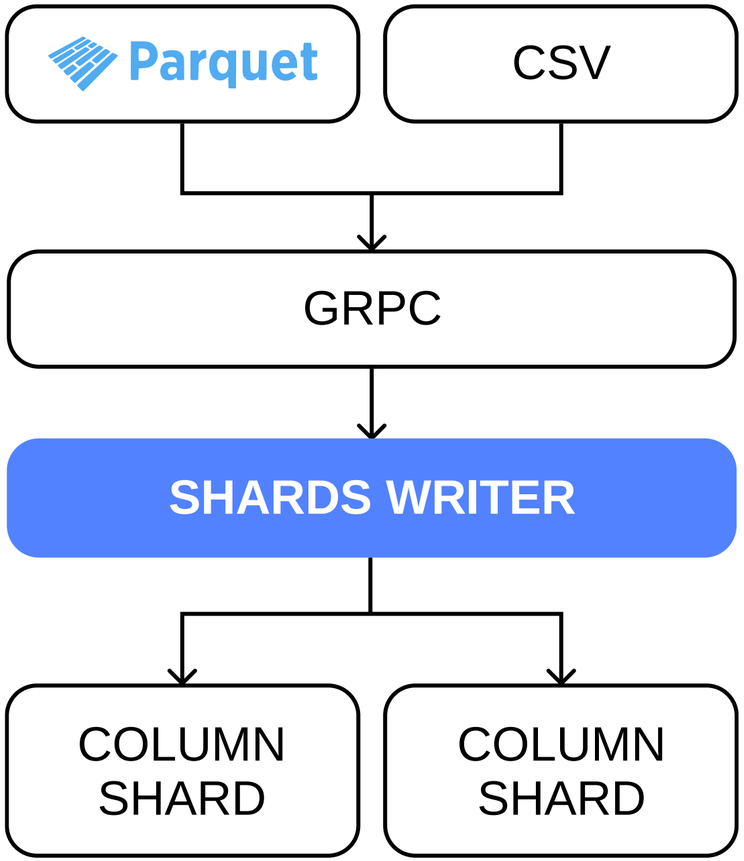
- В gRPC приходят данные в формате csv/parquet



Заливка данных

YDB

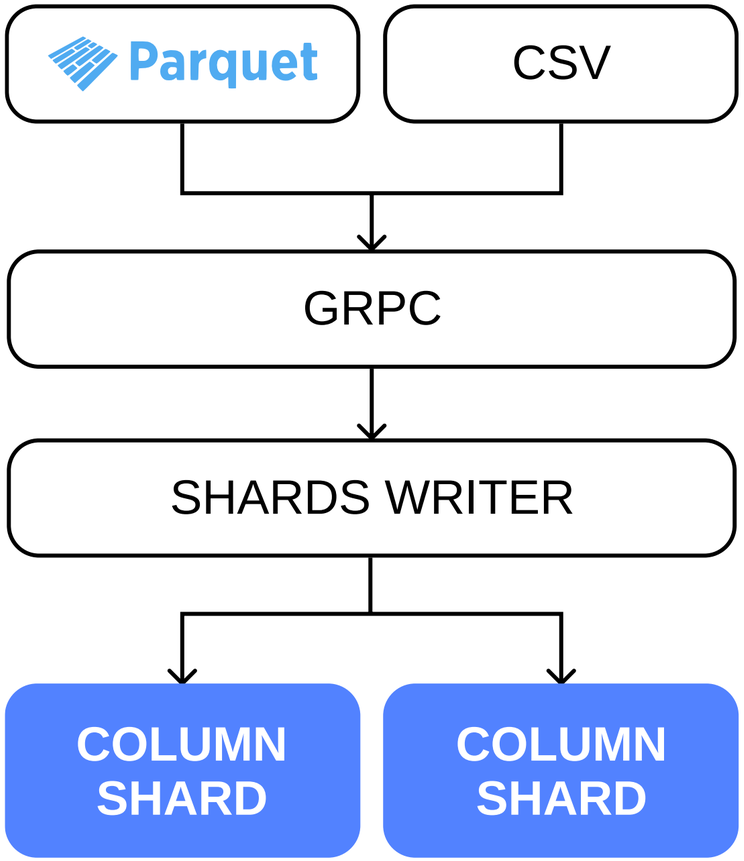
- В gRPC приходят данные в формате csv/parquet
- Шардируются по Primary Key



Заливка данных

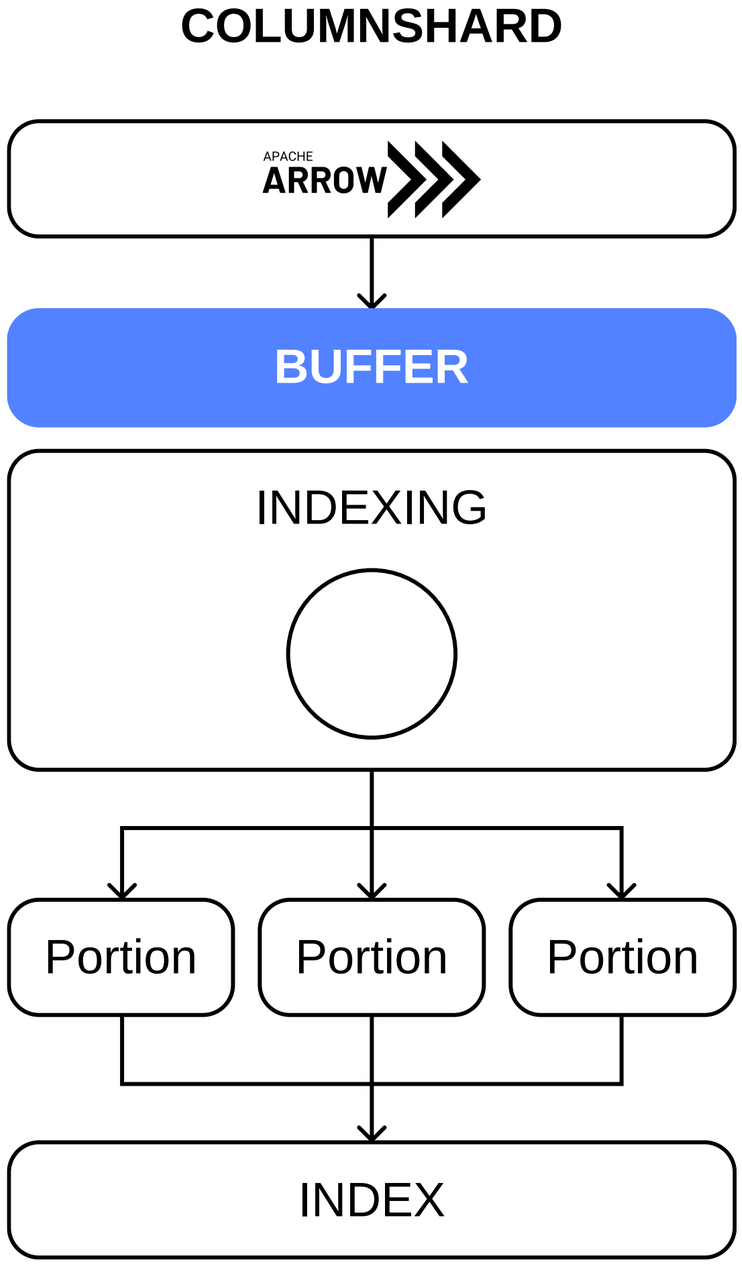
YDB

- В gRPC приходят данные в формате csv/parquet
- Шардируются по Primary Key
- Улетают в таблетки (шарды)



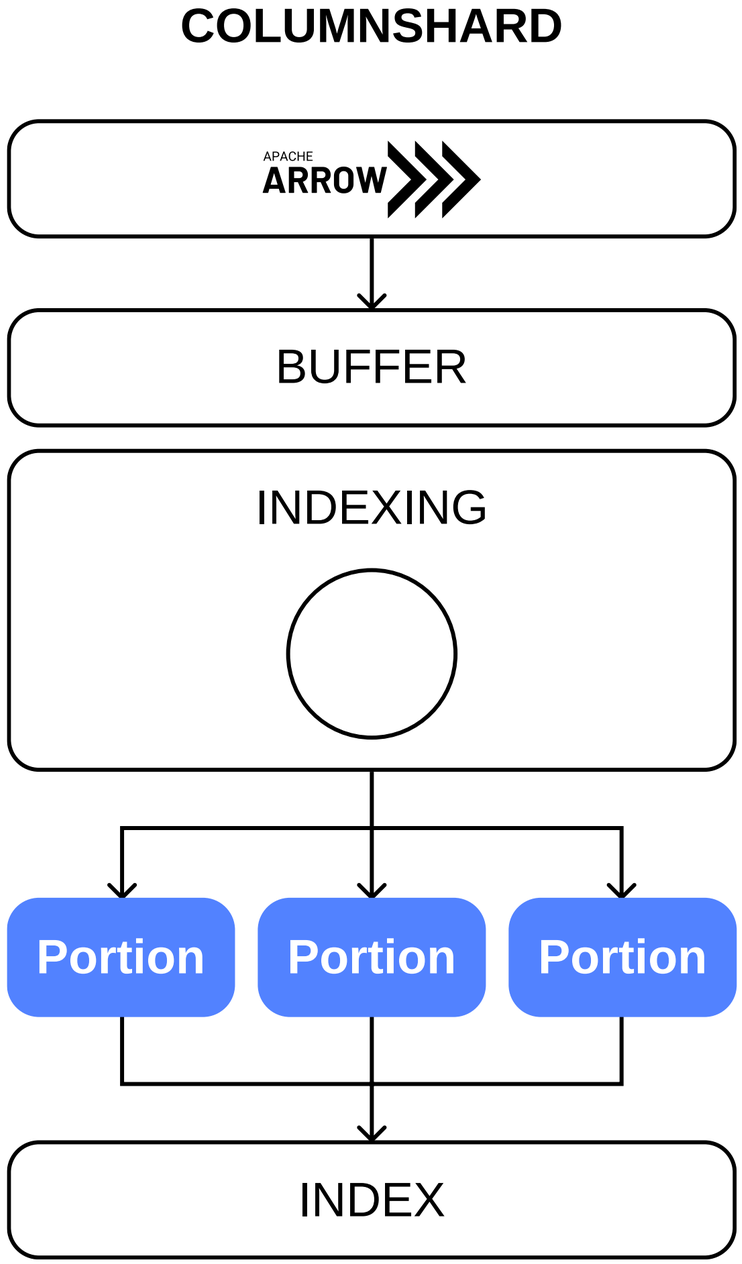
Заливка данных

- В gRPC приходят данные в формате csv/parquet
- Шардируются по Primary Key
- Улетают в таблетки (шарды)
- Оседают в промежуточном буфере



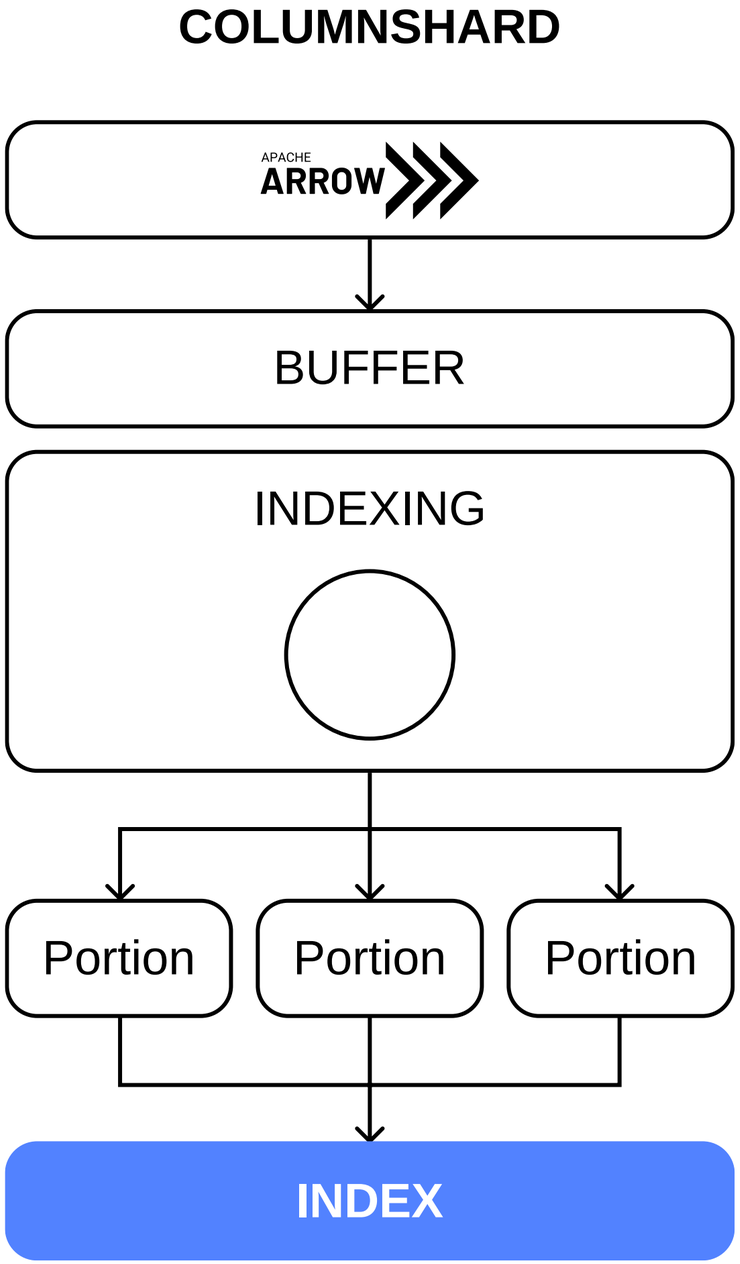
Заливка данных

- В gRPC приходят данные в формате csv/parquet
- Шардируются по Primary Key
- Улетают в таблетки (шарды)
- Оседают в промежуточном буфере
- Регулярным процессом уносятся и перепакуются в порции



Заливка данных

- В gRPC приходят данные в формате csv/parquet
- Шардируются по Primary Key
- Улетают в таблетки (шарды)
- Оседают в промежуточном буфере
- Регулярным процессом уносятся и перепакуются в порции
- Оседают в индекс



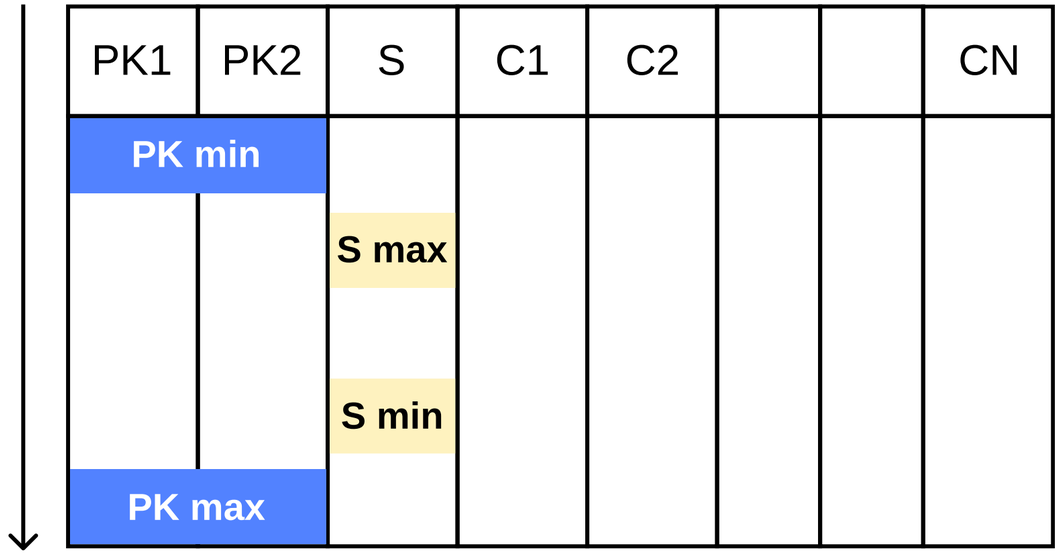
Порция

Что это?

Порция

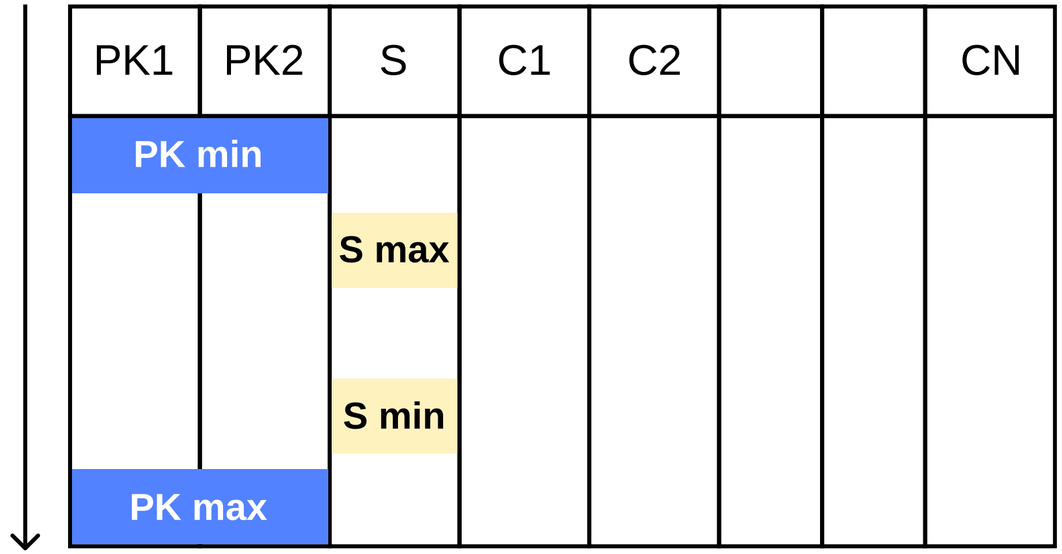
Аналог Sorted Strings Table.

Фрагмент таблицы
из N строк, который
содержит все колонки.



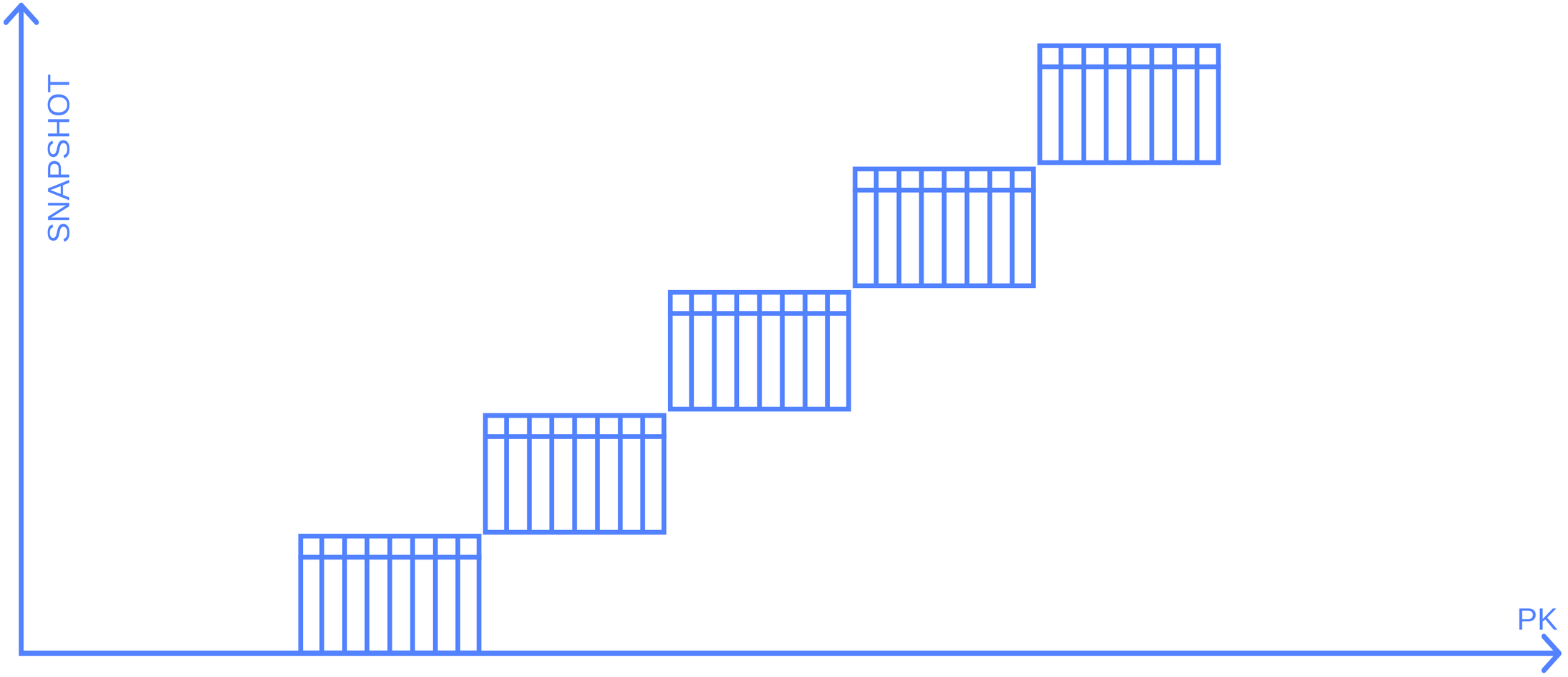
Свойства порции

- Отсортирована по PK
- Каждая строка в порции имеет свой Snapshot (время фиксации в базе)
- Порции могут пересекаться по интервалам PK
- И по интервалам Snapshot



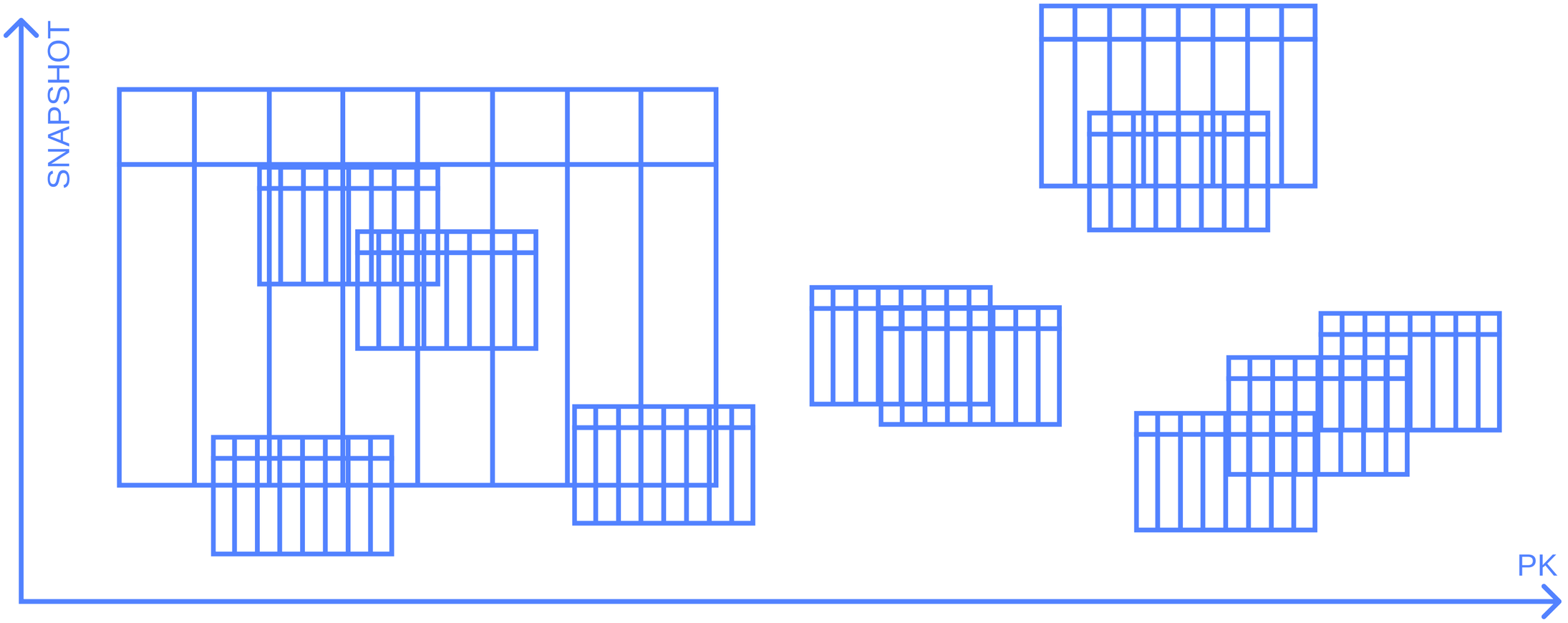
Как порции формируют индекс

Примерно так:

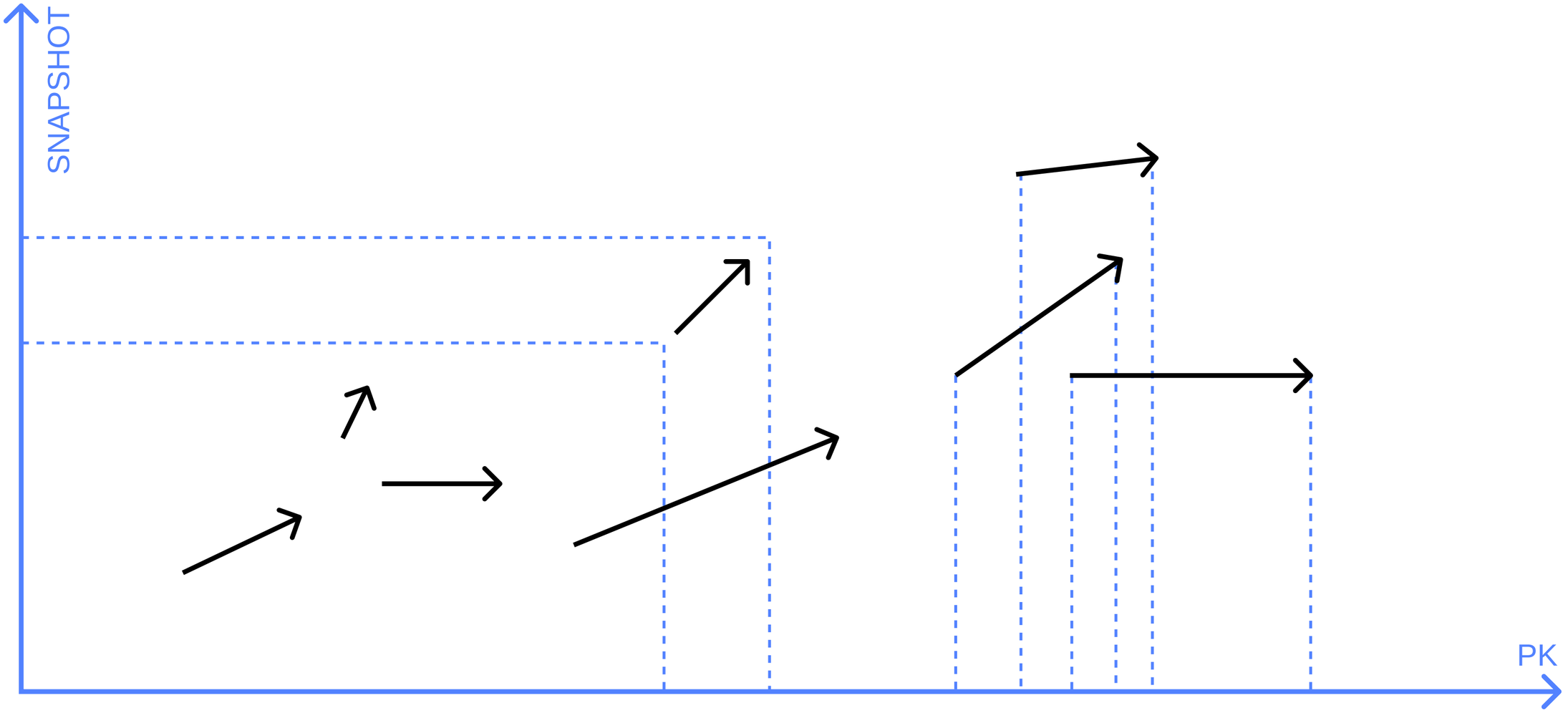


Как порции формируют индекс

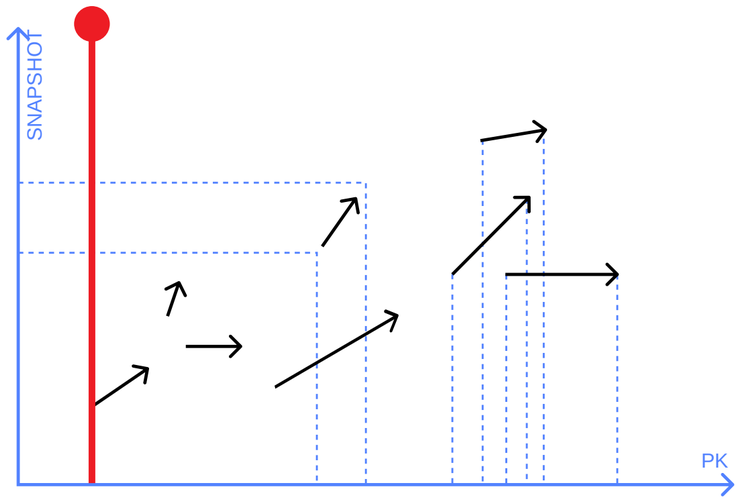
На самом деле вот так:

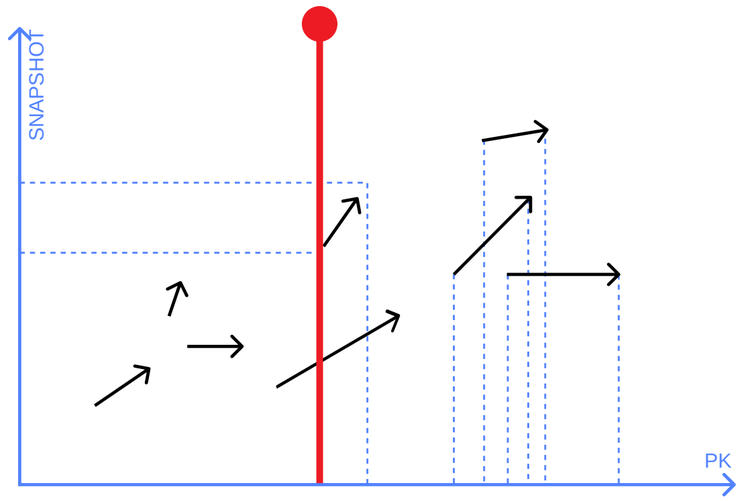


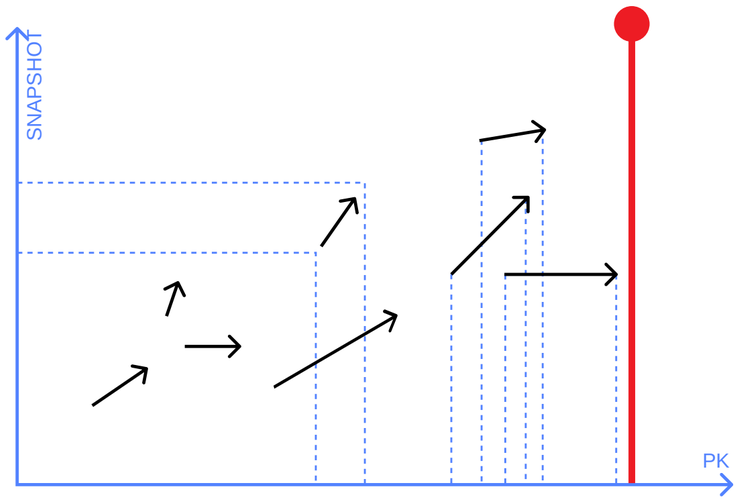
Как порции формируют индекс



Как происходит чтение

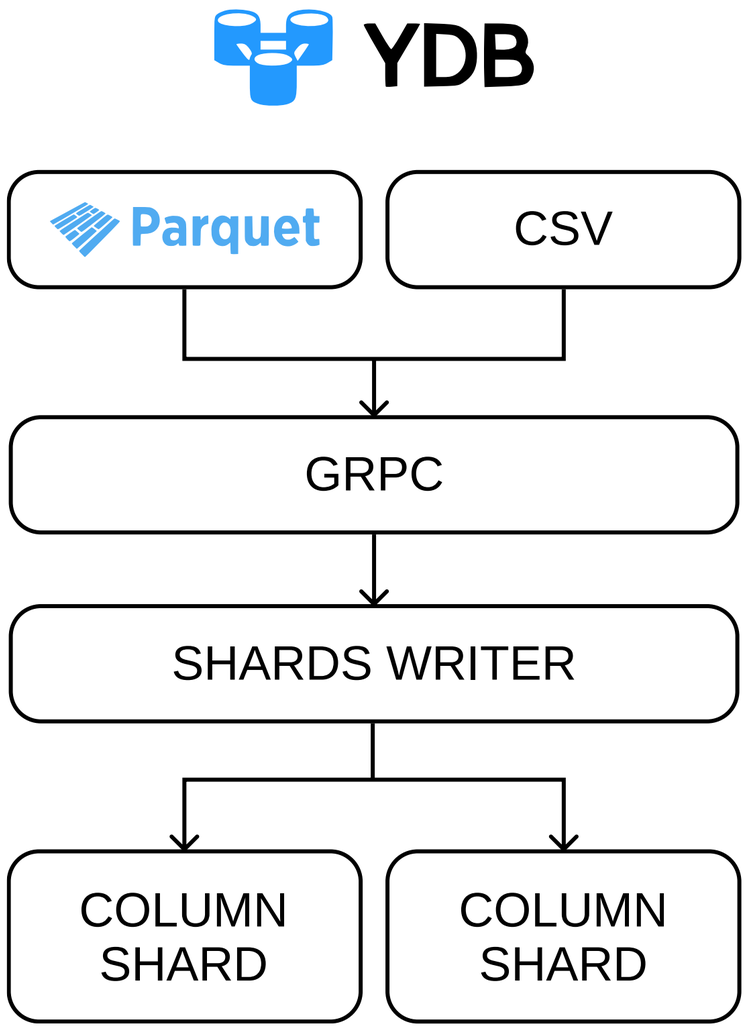


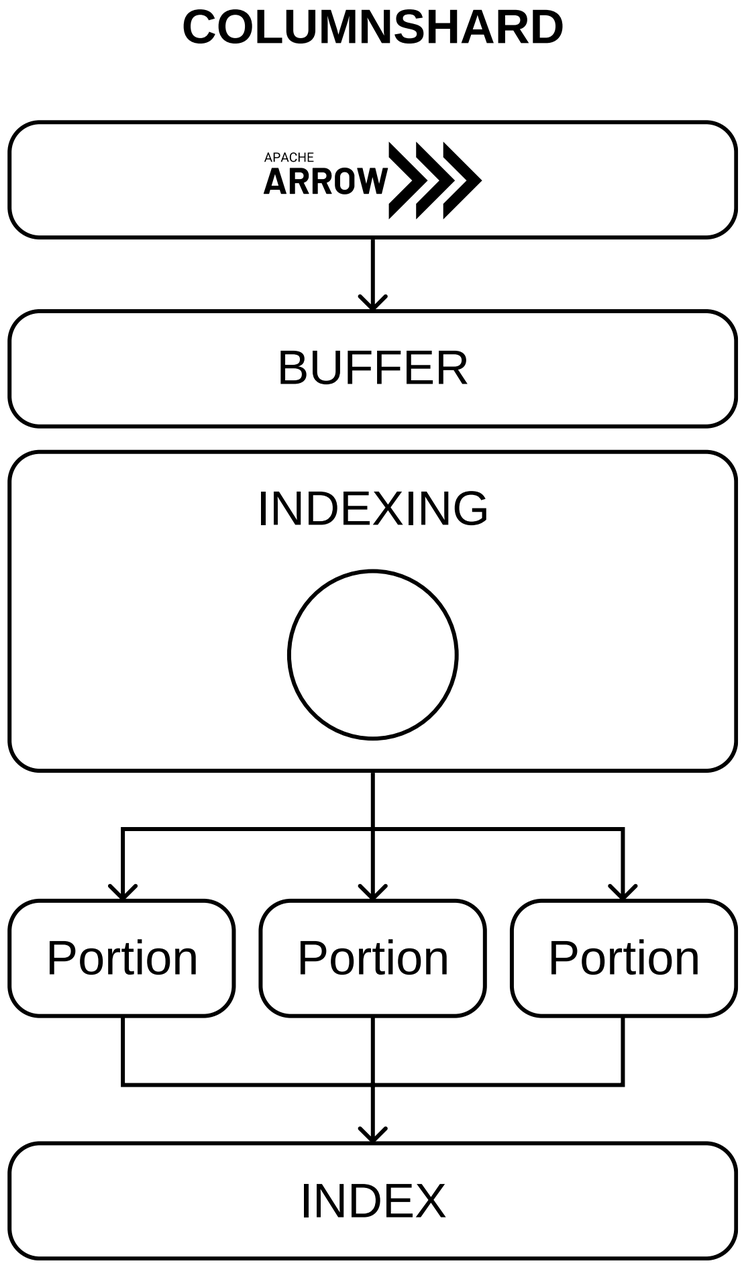




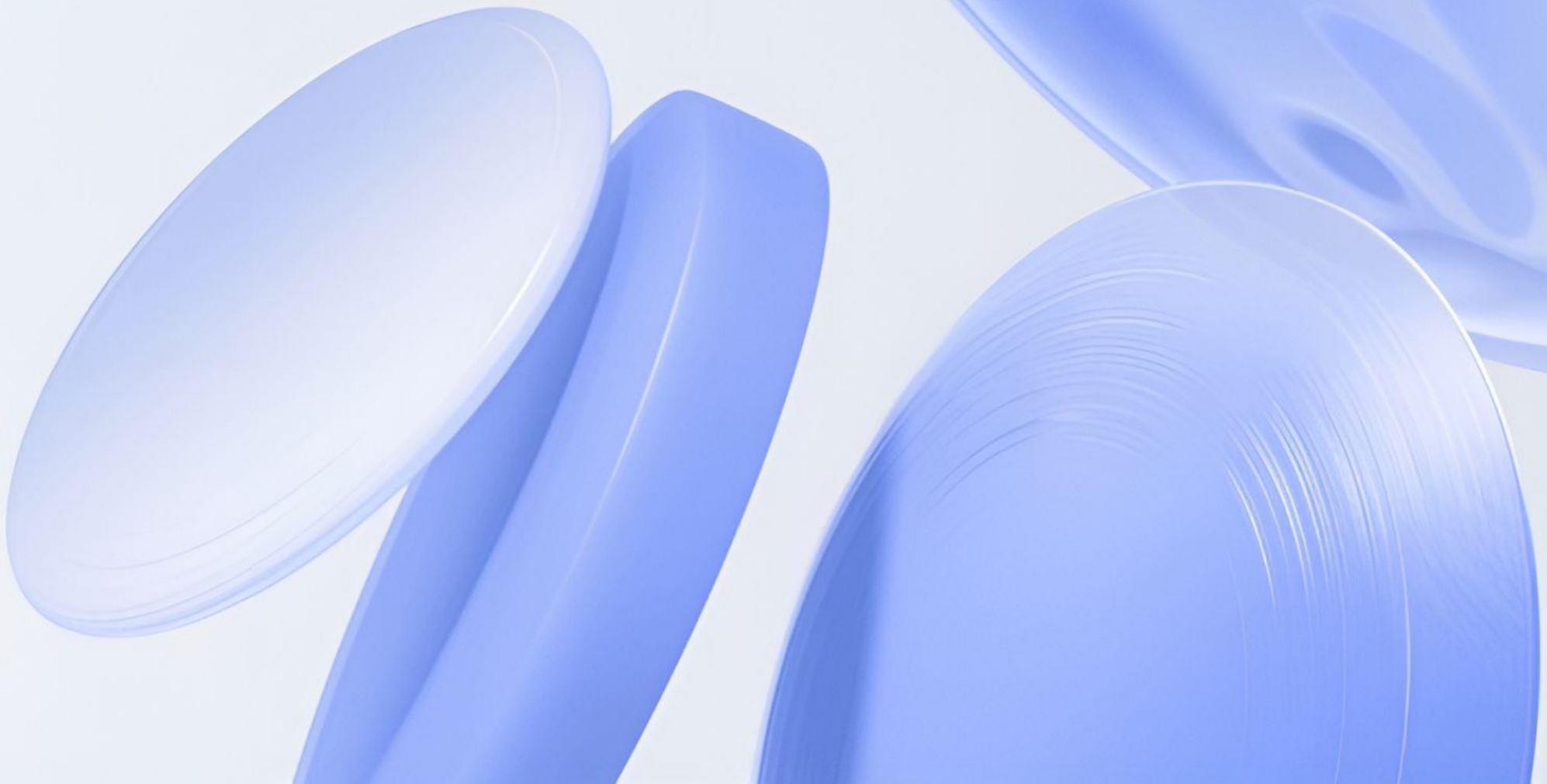

```
SELECT * FROM TABLE
```

Заливка данных





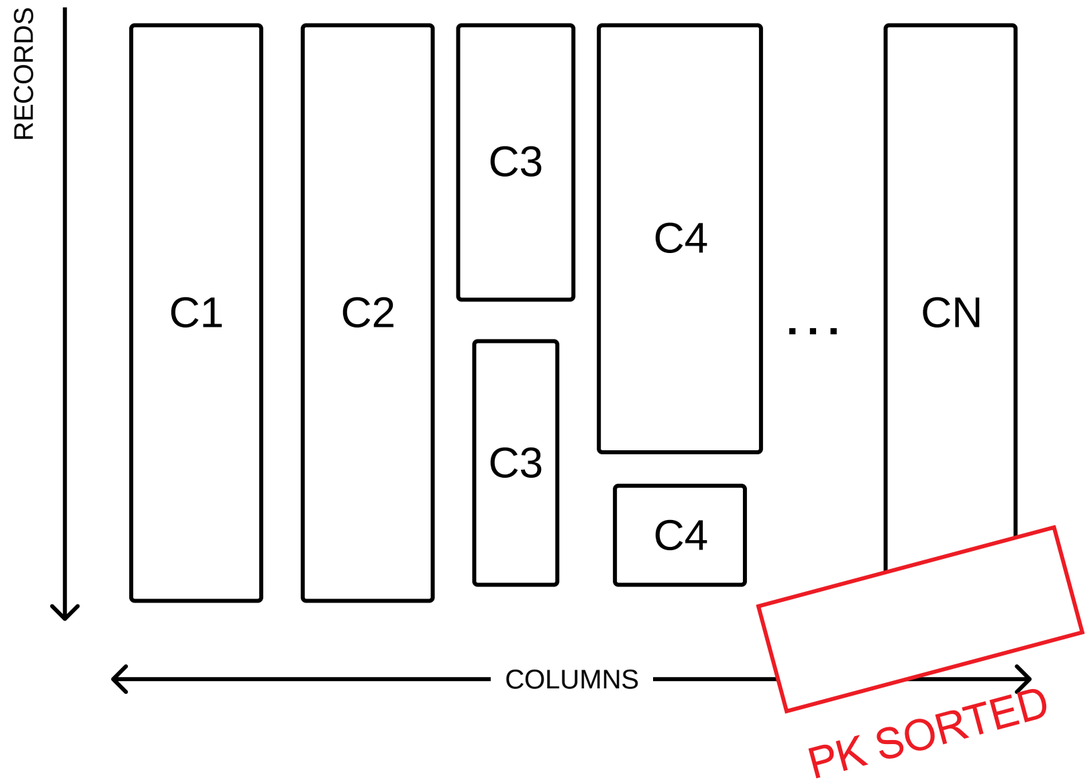
Упаковка данных в порцию



HighLoad++

Порция – колонка – чанк

- Данные бьются на порции
- Порция – это набор колонок
- Колонка может быть физически разбита на K чанков
- Чанки упаковываются согласно политике своей колонки
- Далее объединяются в блобы и сохраняются на диск



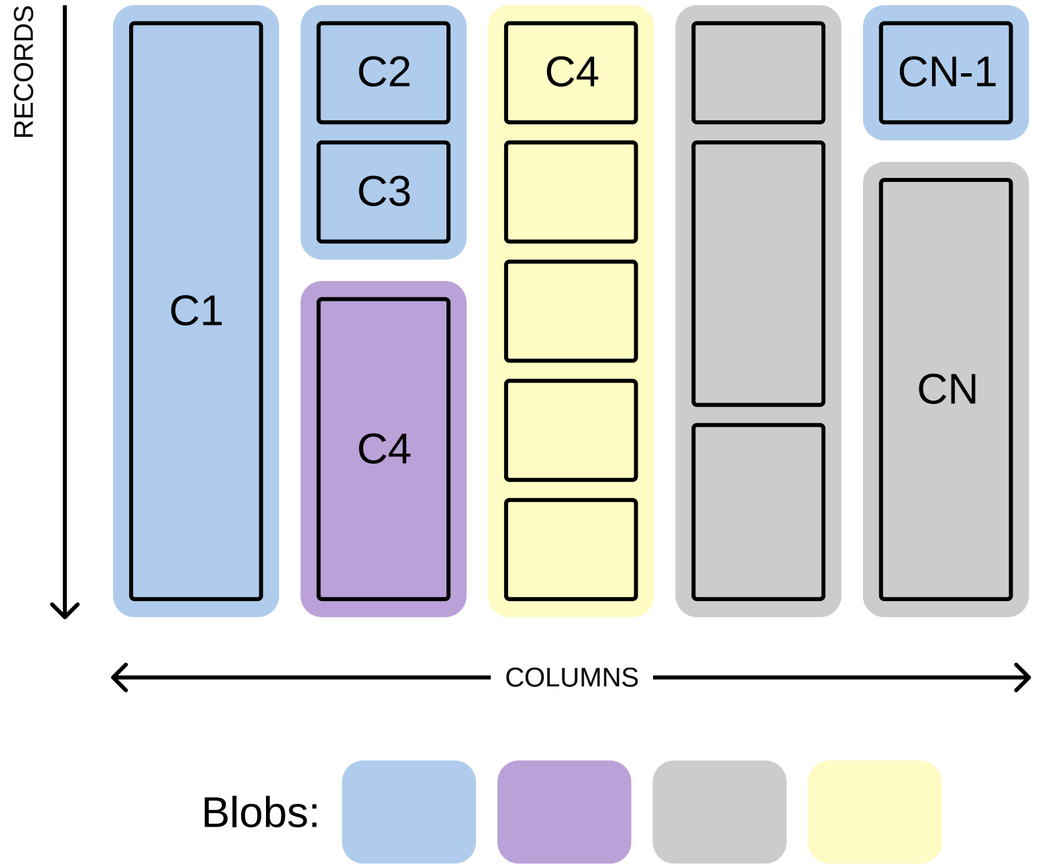
Порция – колонка – чанк

Можем упаковать чанки мелких колонок в один блоб;

Пишем сжатые данные. Поэтому:

- Сжатый чанк должен помещаться в блоб;
- Распакованный – в память.

Круто, если при перепаковке данных получится переносить чанки колонок целиком, не распаковывая.

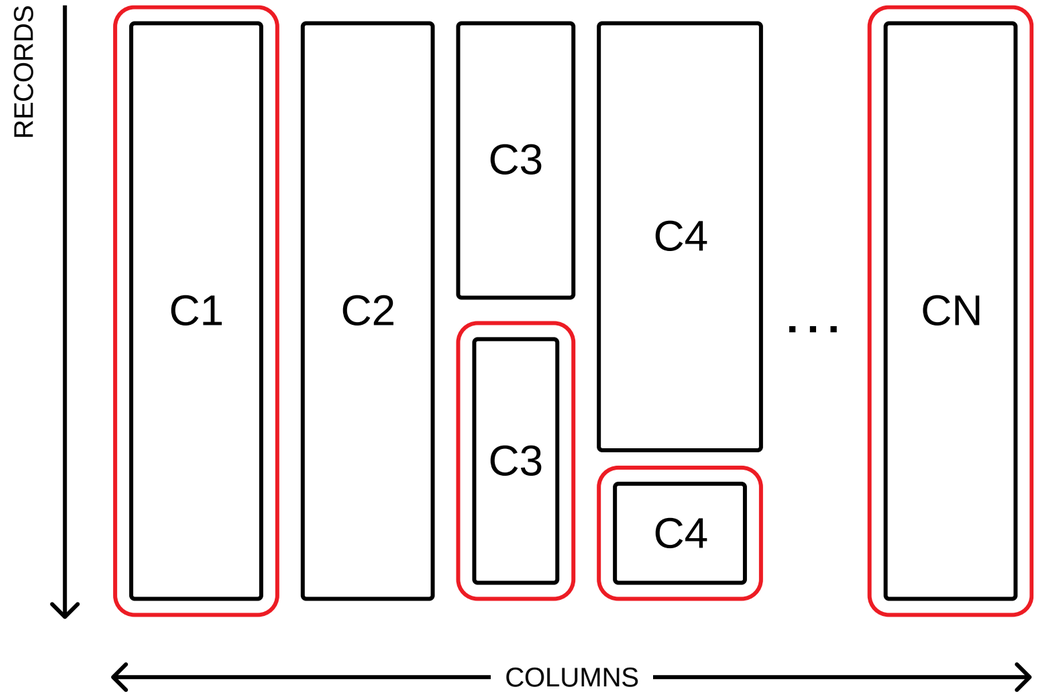


Вариант алгоритма упаковки

- Входящие данные режем на сегменты (определяем границы порций).
- Внутри сегмента сортируем колонки по размеру упакованных данных.
- Фиксируем ограничения на минимальный и максимальный размер блоба.
- Набираем колонки в блоб жадно, пока не упремся в размер.
- Если можем уложить колонку в чанк, не разрезая – не режем.

Чтение чанков

- Данные бьются на порции
- Порция – это набор колонок
- Колонка может быть физически разбита на K чанков
- Чанки упаковываются согласно политике своей колонки
- Далее объединяются в блобы и сохраняются на диск



Как оценивать эффективность упаковки?

Наблюдать количество:

- разрезаемых чанков при упаковке;
- перепаковываемых байт при разрезании чанков;
- мелких блобов в сторадже;
- чанков в сравнении с количеством блобов;
- чанков на колонку (min/max/avg);

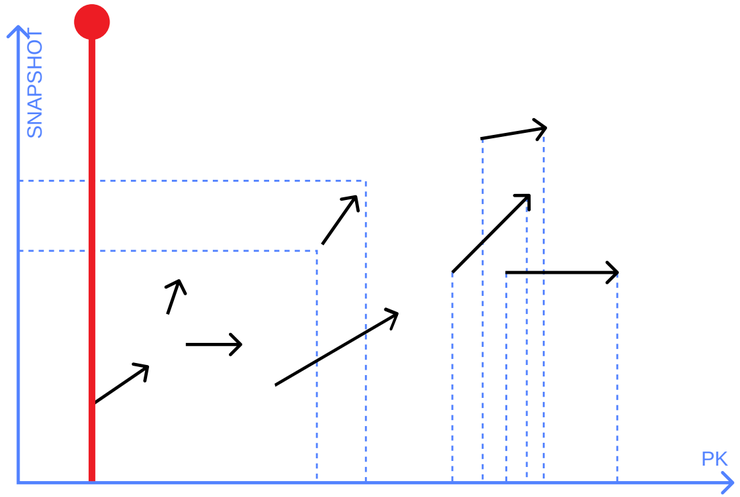
Наблюдать коэффициент сжатия колонки
в зависимости от объема данных в ней.

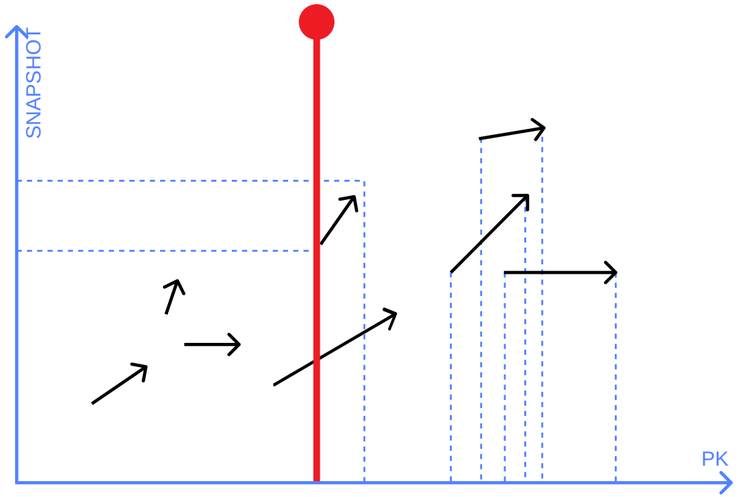
Чтение данных из большого числа порций

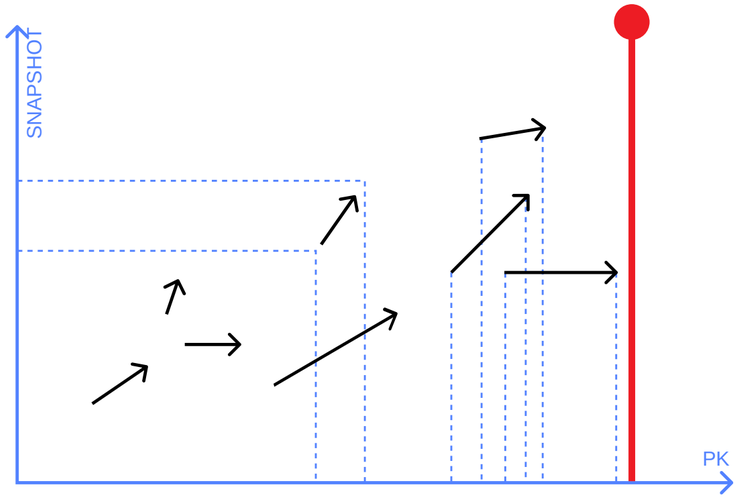
HighLoad++

Сканирование таблицы

- Много маленьких порций – слишком большой индекс.
- Мало больших порций – нужно много данных вытаскивать при чтении по ключу.
- Если порции пересекаются, приходится мерджить







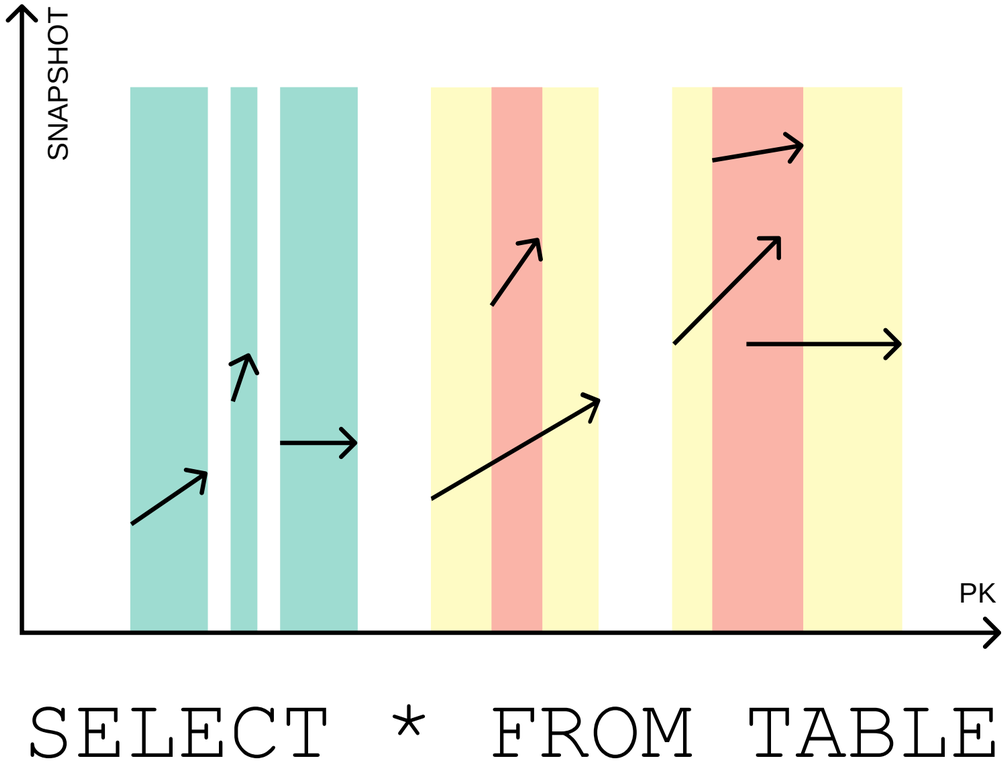
Сканирование таблицы

Чем больше пересекающихся порций,
тем медленнее идет сканирование.

— прочитал и отдал
целиком в ответ O_1

— на заданном интервале можно
отдавать блоки строк,
не проверяя пересечения O_{logN}

— необходимо распаковать ключи
и склеить по снапшоту O_N

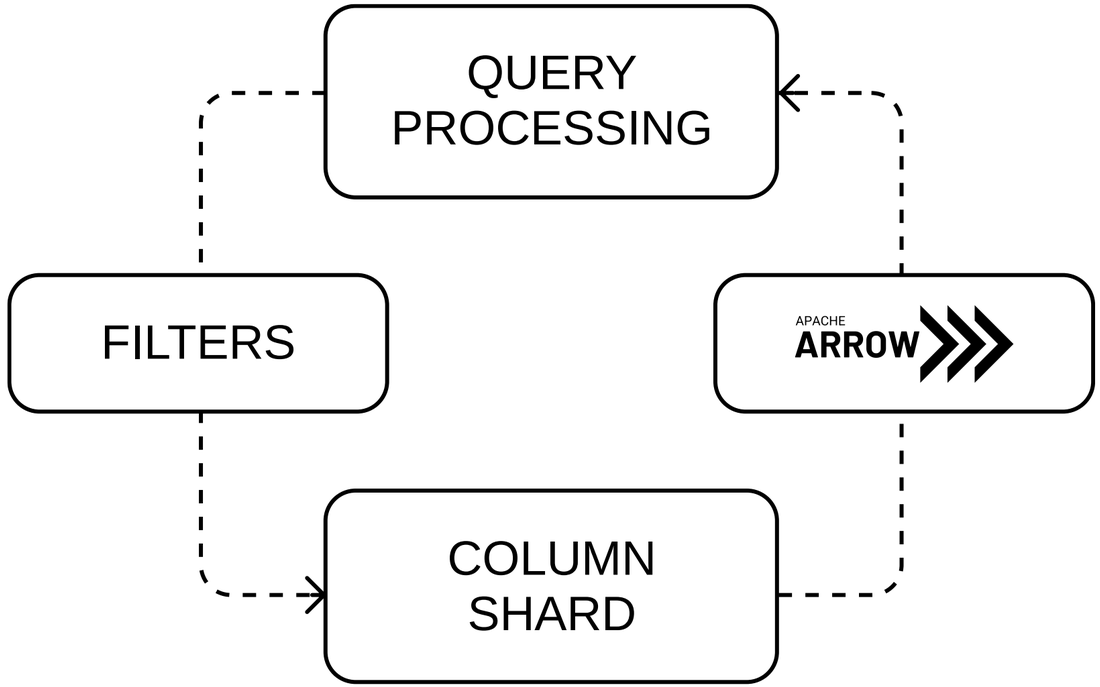


Фильтры

Идея: для оптимизации поиска мы спускаем в шарды часть фильтров.

Фильтры применяются в процессе сканирования перед слиянием.

Это позволяет нам делать дополнительные оптимизации.



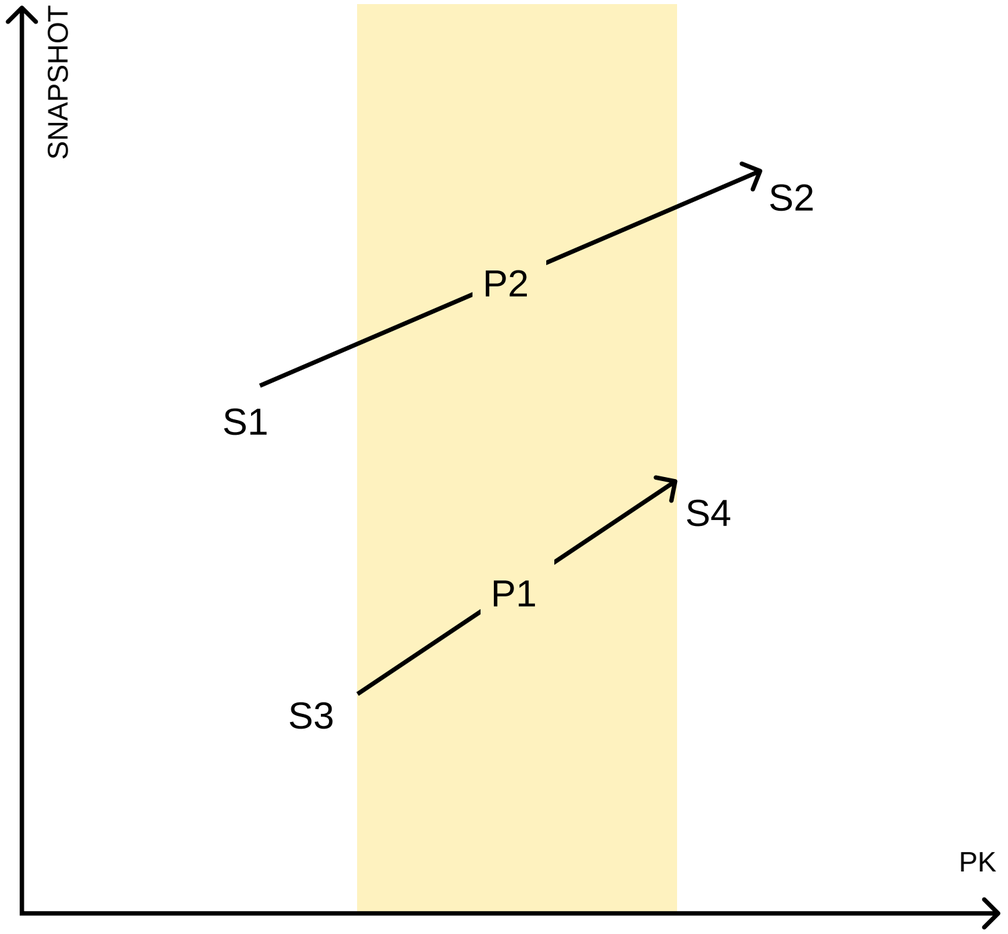
PK1	PK2	S	C1	C2			CN

Оптимизация по снапшоту

Если

$$S_1 > S_4 \&\& \text{Filtered}(P_1) == \varnothing$$

то P_1 можно просто выкинуть
и не делать мердж

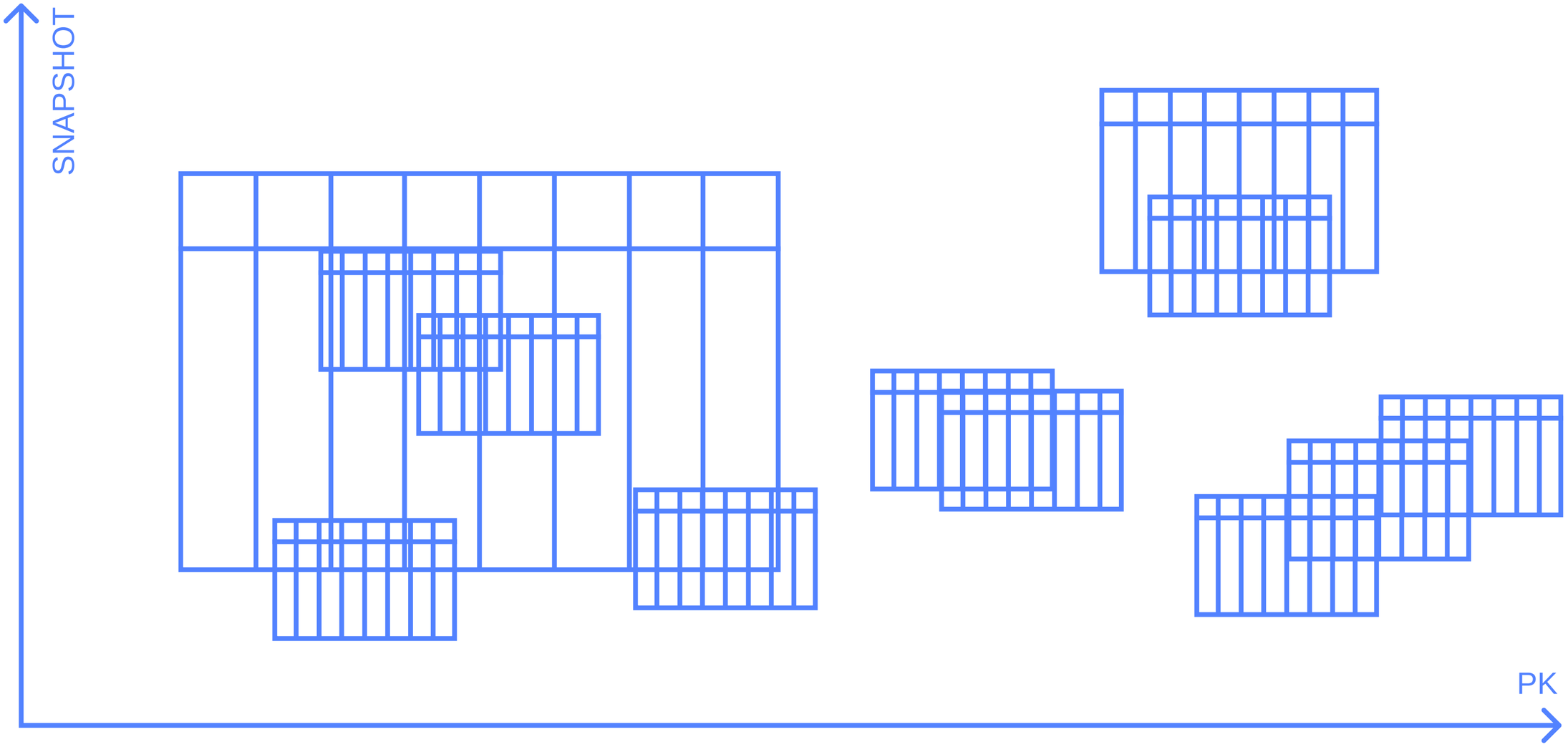


Вспомогательные конструкции

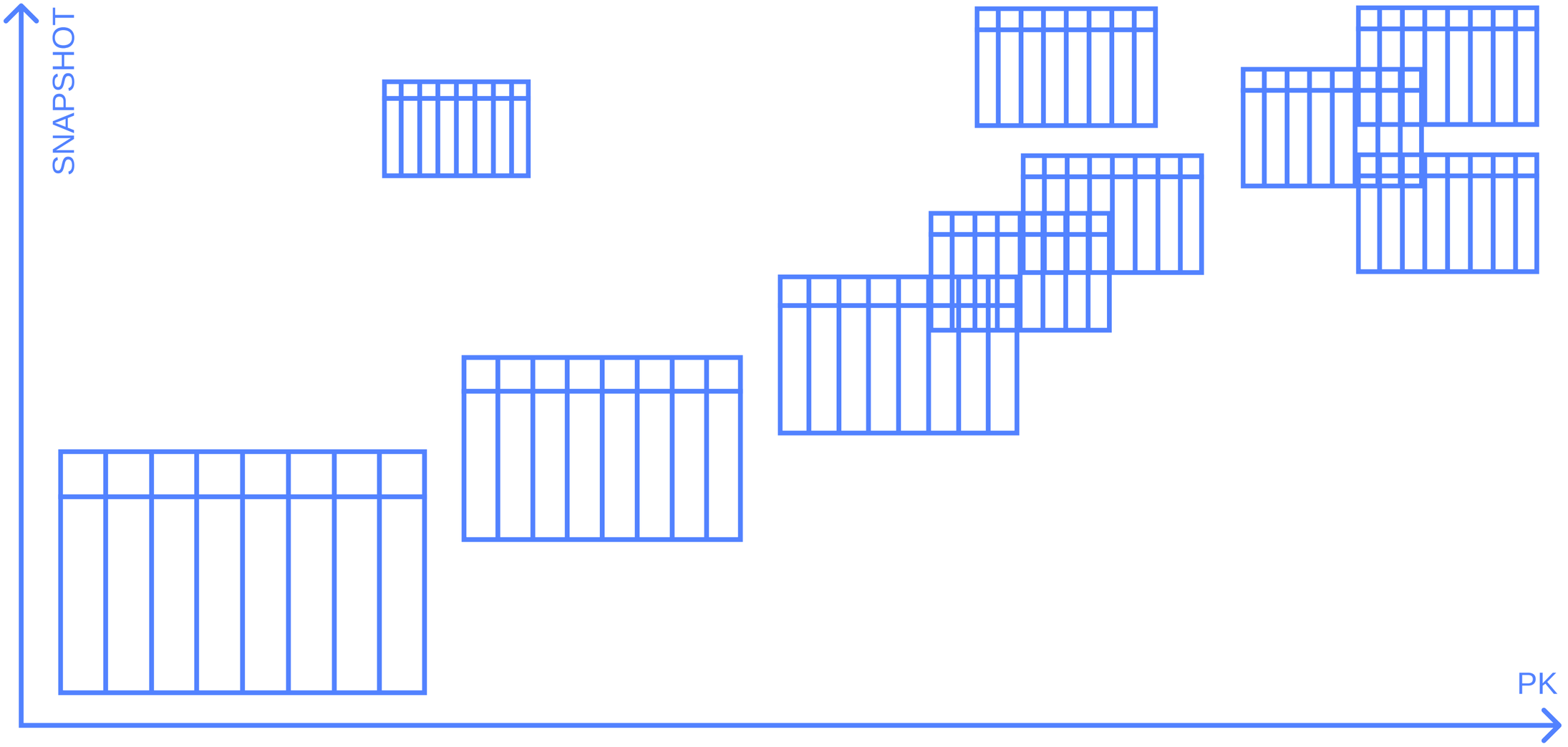
```
// Задача на чтение данных
void StartScan() {
    NResourceBroker::NSubscribe::ITask::StartResourceSubscription(ResourceSubscribeActor,
        std::make_shared<TReadPortionsTask<TConveyorTask>>(blobIds),
                                                    cpuCount, memSize,
"CS::SCAN");
}

// Обработка результатов в конвейере
virtual void DoOnDataReady() override {
    std::shared_ptr<NConveyor::ITask> task =
                                std::make_shared<TConveyorTask>(ExtractBlobsData(),
Schemas);

    NConveyor::TCompServiceOperator::SendTaskToExecute(task);
}
```

Оптимизация индекса

Плохое состояние



Хорошее состояние



Compaction = Optimizer + Repack

Оптимайзер

- Поддерживать актуальность метрик оптимизации;
- Выдавать задачи для упаковщика за O_1;
- Минимизировать количество перезаписей данных.

Задача

- Список порций;
- Check points — гарантированные границы результирующих порций;
- Различные лимиты для построения результата.

Compaction = Optimizer + Repack

Задача

- Список порций;
- Check points — гарантированные границы результирующих порций;
- Различные лимиты для построения результата.

Упаковщик

- Контролирует потребляемую память (unpacked_bytes может оказаться большим);
- Загружает в память только необходимые для текущего мерджа чанки.

Compaction = Optimizer + Repack

Упаковщик

- Контролирует потребляемую память (unpacked_bytes может оказаться большим);
- Загружает в память только необходимые для текущего мерджа чанки.

Результат

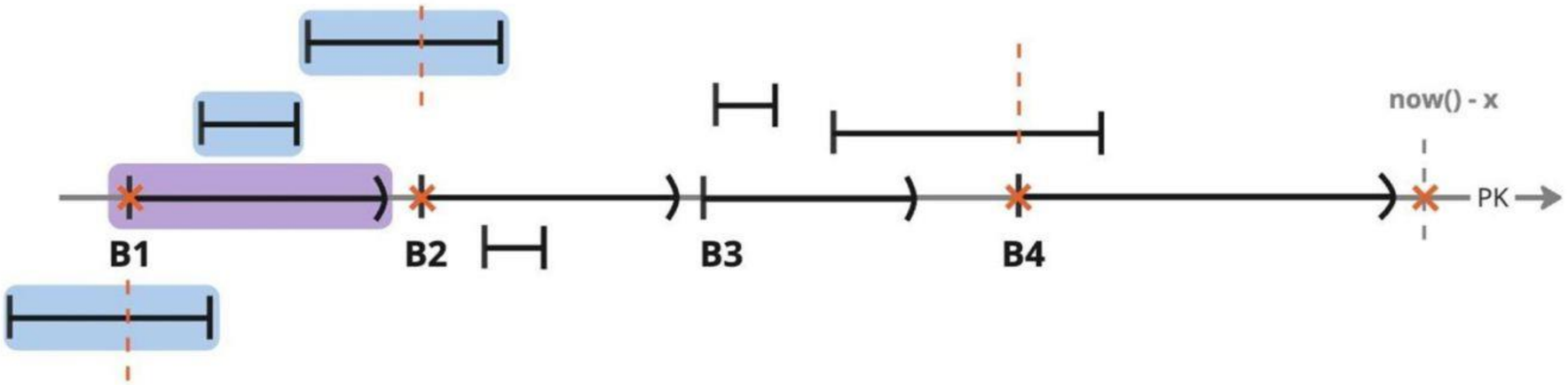
- Порции не пересекаются;
- Порции сортированы по PK;
- Порции разделены check point’ами
- Все неактуальные данные удалены физически.

Свойства оптимайзера

- Изолированность. Дает возможность реализовывать и сравнивать разные подходы к оптимизации.
- Адаптируется под профиль заливки.
- Имеет гарантированный объем ресурсов.
- Использует с низким приоритетом ресурсы, недоиспользованные на поиске.
- Оптимизирует целевую функцию F (мелкие порции, пересекающиеся интервалы, …).

Режим оптимайзера — логи

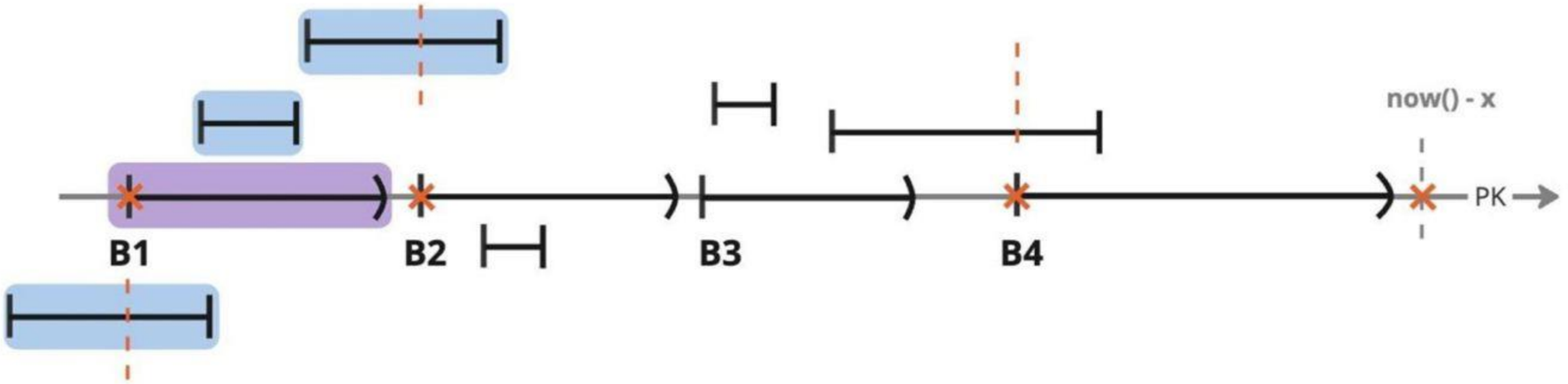
- Делим индекс на бакеты. Хотим, чтобы бакет состоял из одной образующей порции;
- Образующие порции не пересекаются.



Режим оптимайзера — логи

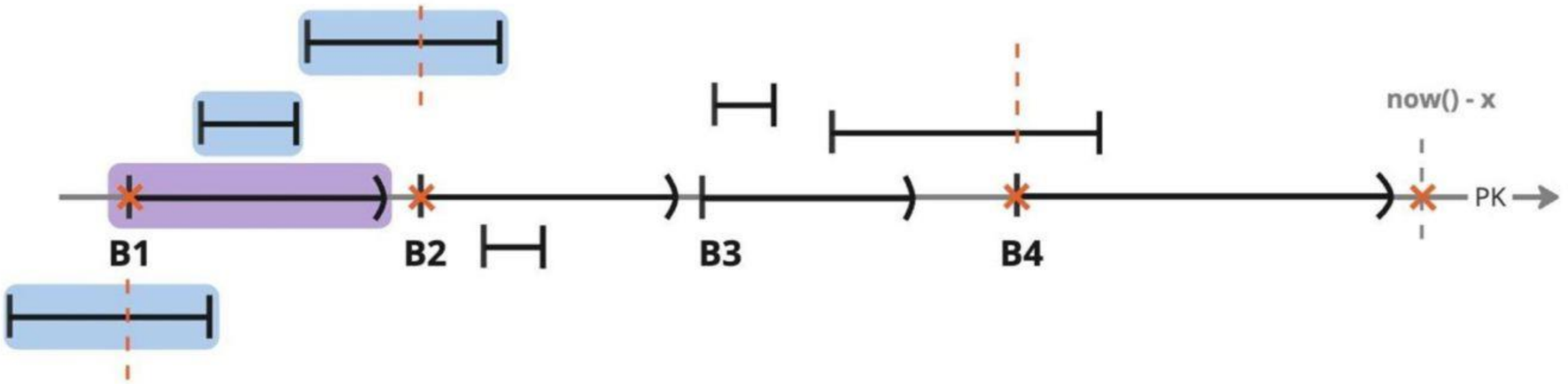
При добавлении новой порции, определяем её как образующую, если:

- Немаленькая;
- Несвежая;
- Не пересекает другие образующие порции.



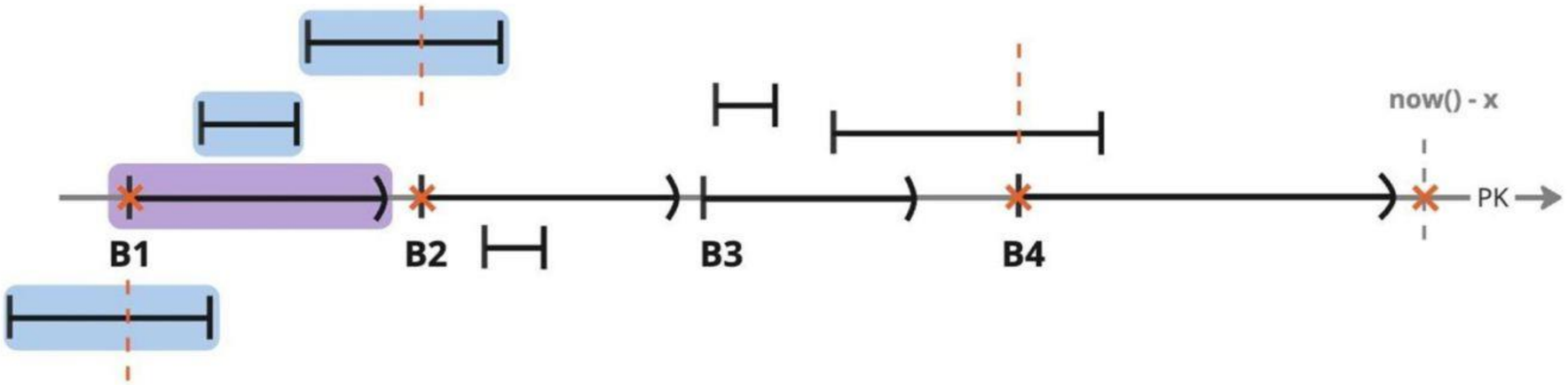
Режим оптимайзера — логи

- Одна задача оптимайзера работает с одним бакетом;
- Бакеты-кандидаты на оптимизацию ранжируются по некоторой формуле.

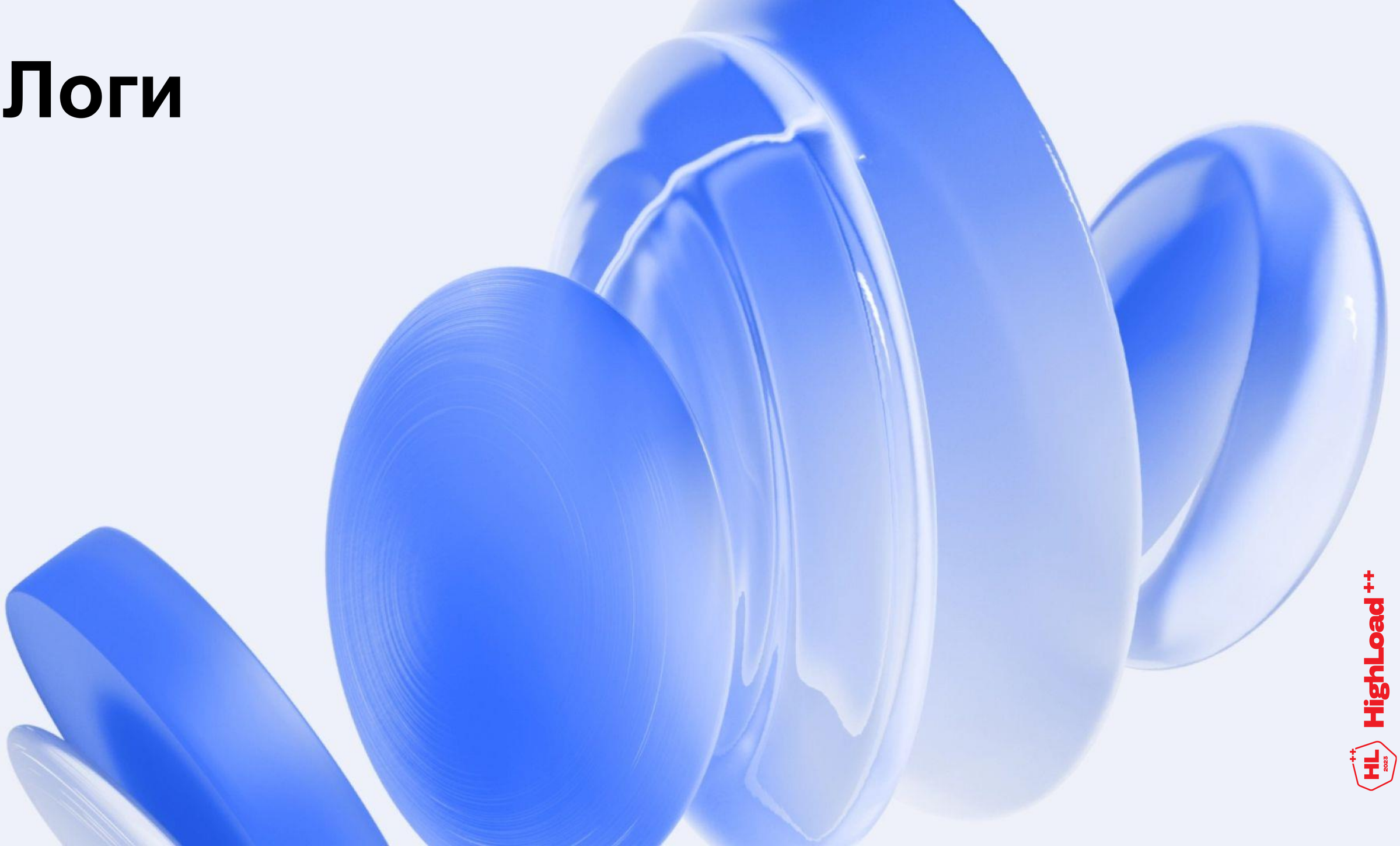


Режим оптимайзера — логи

- Рост от количества мелких порций;
- Рост от количества пересекающихся порций;
- Падение от объема данных, которые придется перепаковать при мердже порций.



Логи



HighLoad++

Конфигурация кластера

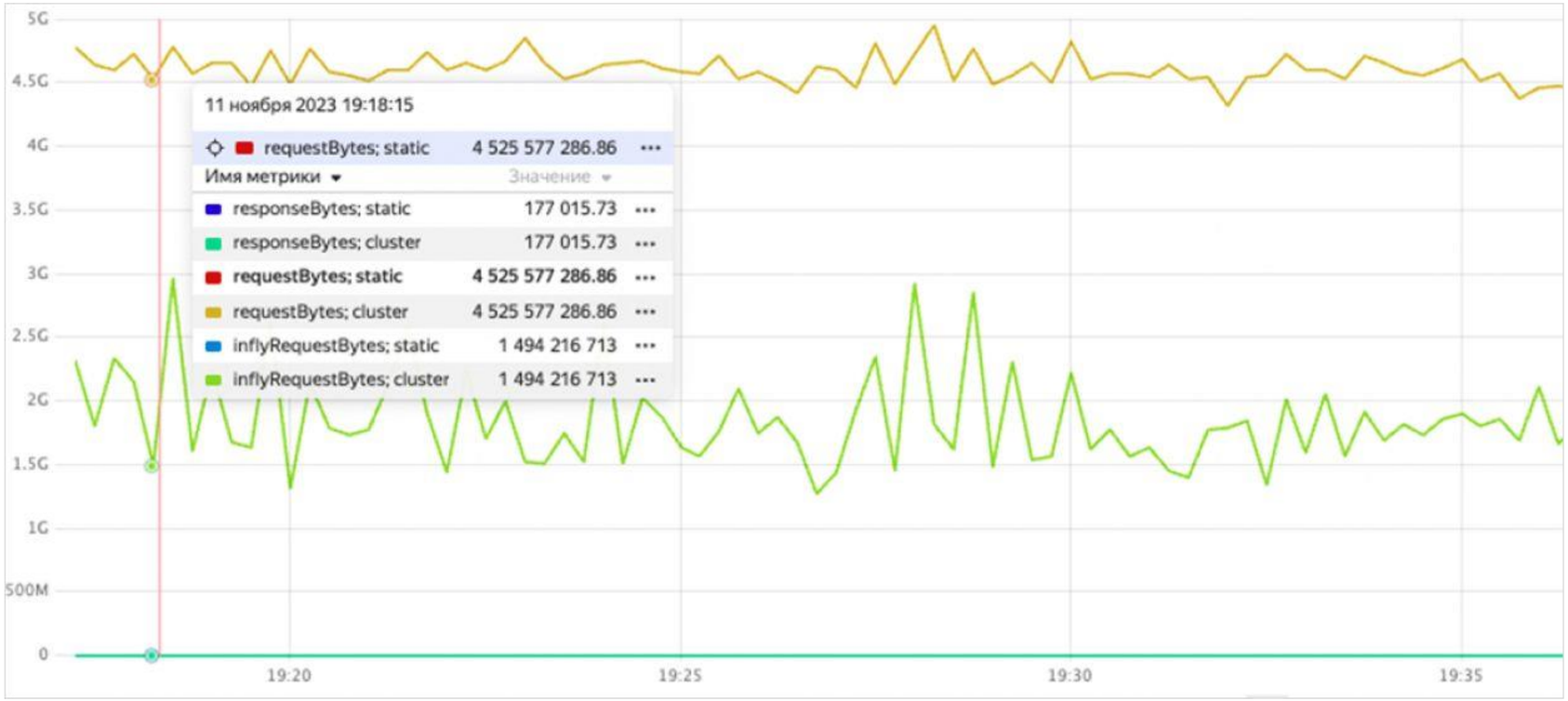
8
хостов

16
вычислительных нод
(по 2 на хост)

16
ядер на ноду

Итого: 256 ядер

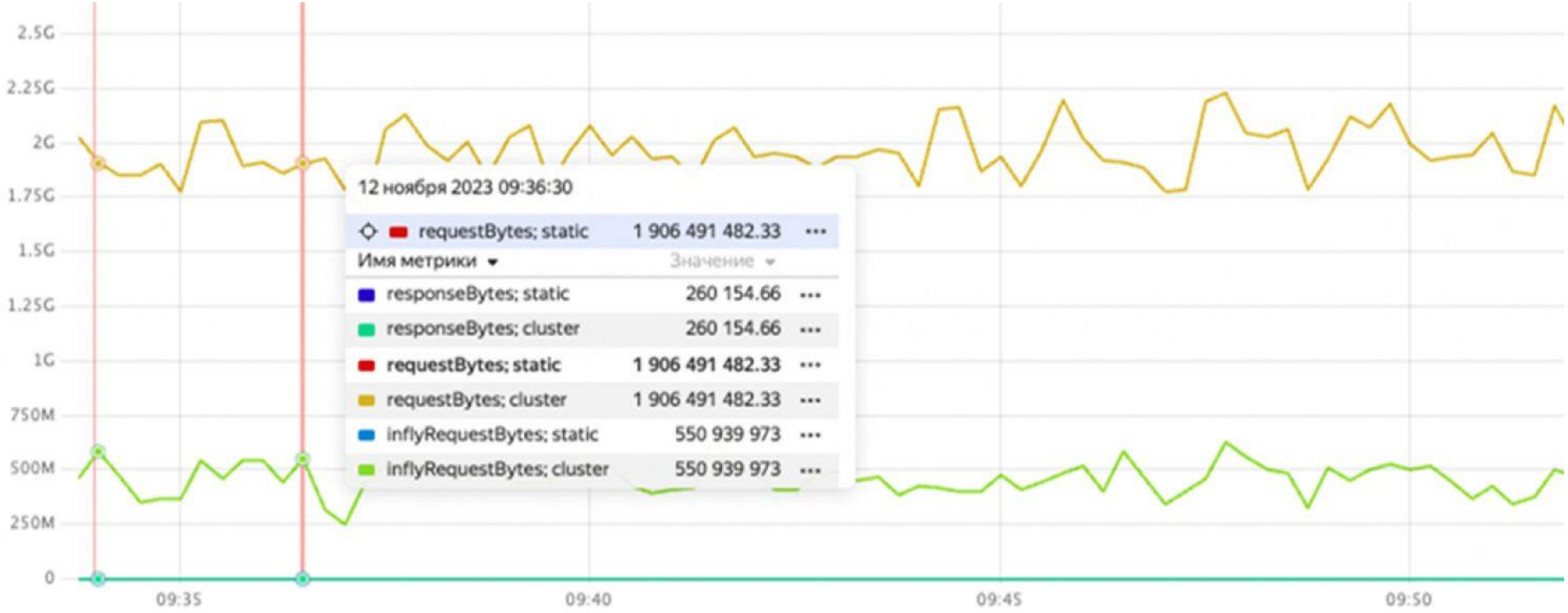
Заливка: до 4,5 Гб/сек



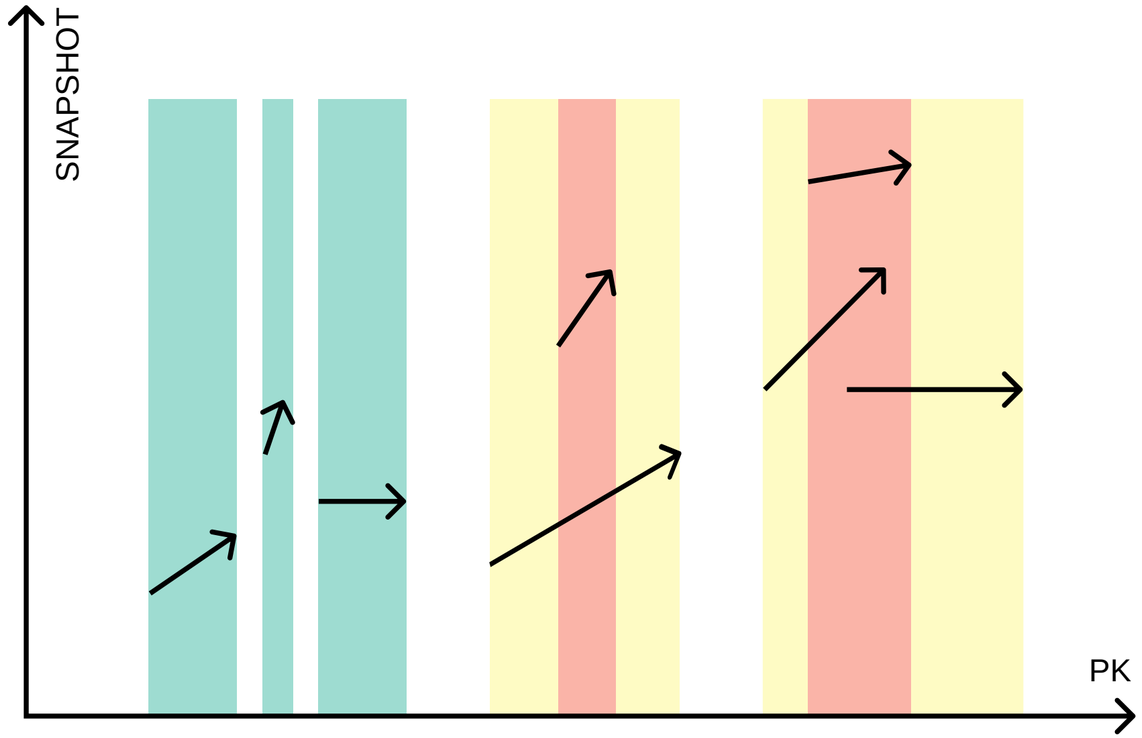
Тестирование поиска

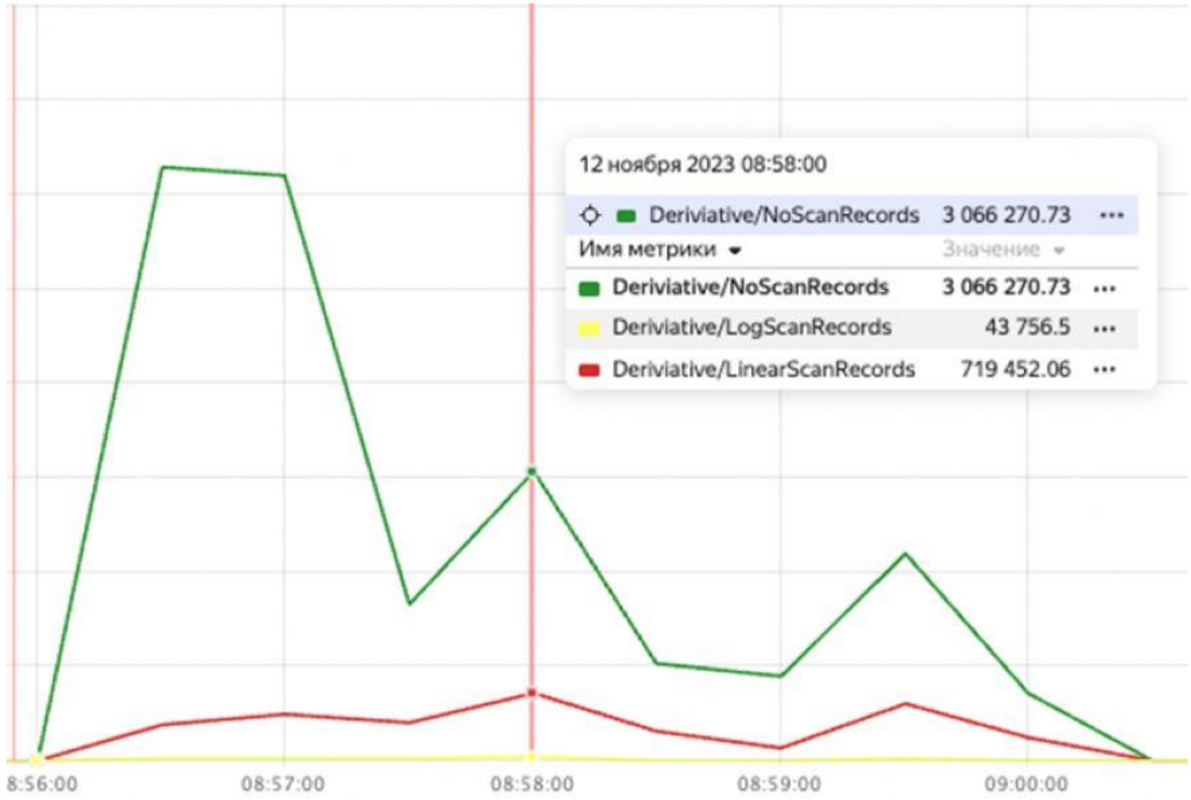
Подбираем интервал так, чтобы в него попало 70 млн записей.

Идет регулярная заливка до 2 Гб/сек.



Тестирование поиска





```
Q0: SELECT COUNT(*), MIN(timestamp), MAX(timestamp)
```

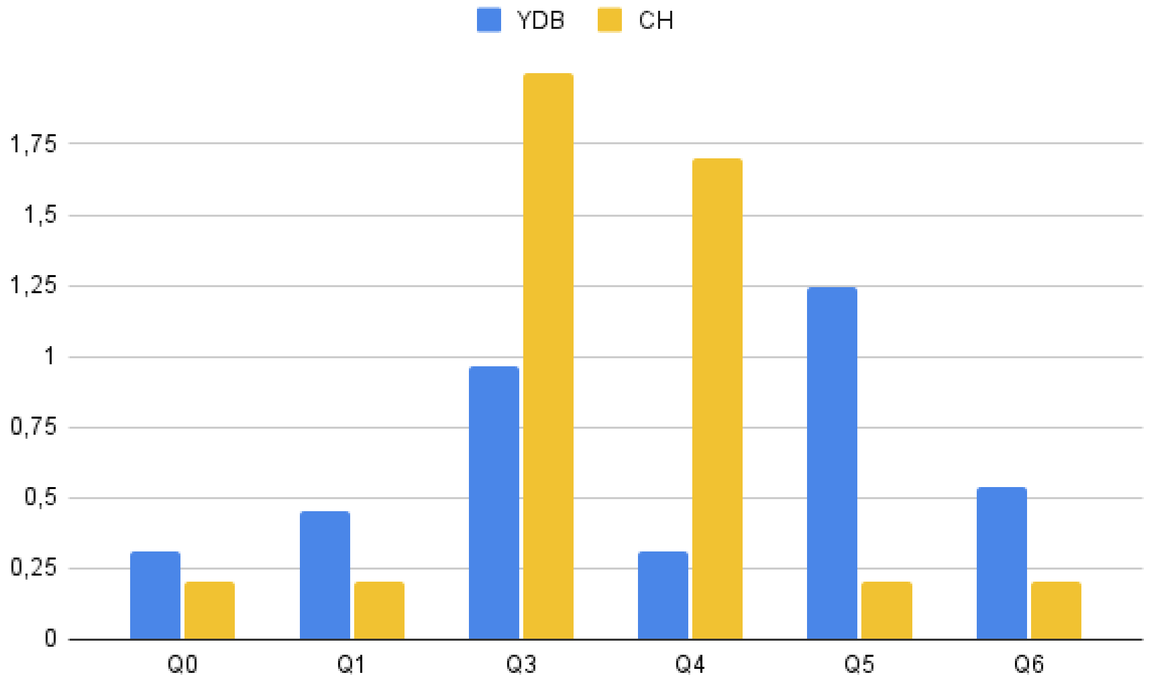
```
Q1: SELECT logger_name, count(*) as count
    GROUP BY logger_name
    ORDER BY count DESC
```

```
Q3: SELECT timestamp, host, seq
    JSON_VALUE(context, "$.'mdc.puid'") = '1345376780'
    ORDER BY timestamp DESC, host DESC, seq DESC
```

```
Q4: SELECT timestamp, host, seq
    message LIKE "%16784436210268789 87%"
    ORDER BY timestamp DESC, host DESC, seq DESC
```

```
Q5: SELECT logger_name, count(*), timestamp
    GROUP BY logger_name, `timestamp`
    ORDER BY count DESC
```

```
Q6: SELECT COUNT(*), MIN(timestamp), MAX(timestamp)
    message LIKE "%16784436210268789 87%"
```



Выводы

- Разработка нового компонента — исследование айсберга;
- Простая идея не всегда означает простоту реализации и скрывает немало нюансов, и это интересно;
- Сложные алгоритмы и неочевидные решения полезно изолировать;
- Вести разработку в рамках существующей платформы выгодно: можно сосредоточиться на задачах предметной области и не думать об инфраструктуре (сеть / хранение / балансировка нагрузки).

Реализовать OLAP

Как мы делали колоночное хранение в YDB

Новожилова Софья, Яндекс

HL++ 2023 HighLoad++