УДК 004.657

© 2020 г. **Ю.А. Григорьев,** д-р техн. наук,
**О.Ю. Ермаков**
(Московский государственный технический университет им. Н.Э. Баумана)

# ОБРАБОТКА ЗАПРОСОВ В СИСТЕМЕ С ЛЯМБДА-АРХИТЕКТУРОЙ НА УРОВНЕ УСКОРЕНИЯ

Выполнен анализ процессов потоковой обработки данных на уровне ускорения, включающий звено сбора данных, звено очереди сообщений, звено анализа, хранилище данных в памяти и звено доступа к данным. Рассмотрен алгоритм Count-Min Sketch для подсчета частоты и суммы значений какого-либо элемента в потоке. Показано, что использование эскиза (sketch) приводит к большой ошибке восстановления накопленных значений при достаточно большом числе элементов в потоке. Предложена реализация звена анализа на уровне ускорения в системе с лямбда-архитектурой с плавающим окном. Звено включает матрицу векторов (одномерных числовых массивов) вместо эскизов. Это позволяет читать накопленные значения из векторов матрицы напрямую.

***Ключевые слова:*** лямбда-архитектура, потоковая обработка, уровень ускорения, эскиз, вектор.

DOI: 10.22250/isu.2020.64.3-16

## Введение

Обработка больших объемов данных в реальном времени является важным требованием в современных высоконагруженных системах. Для решения этой задачи используются системы потоковой обработки данных. Обработка потоков данных нашла применение в различных областях: в поисковых системах, в социальных сетях, а также в системах обнаружения мошенничества в торговых и финансовых системах, в системах контроля состояния оборудования, военных и разведывательных системах [1].

Одним из вариантов реализации потоковой обработки данных является лямбда-архитектура [2]. В [3, 4] представлена реализация лямбда-архитек-

туры для создания бэкэнда обработки данных в Amazon EC2, обеспечивающая высокую пропускную способность при невысокой стоимости обслуживания сети. Лямбда-архитектура используется для реализации потоковой обработки во многих других областях: отслеживание тепловой карты (heatmap) [5], обработка запросов [6], в медицине для внутри хирургических прогнозов [7] и др.

Лямбда-архитектура имеет пакетный уровень и уровень ускорения. На уровне пакетной обработки хранится главная копия массива данных. На основе этих данных формируются пакетные представления (уровень обслуживания), которые обеспечивают быстрый доступ к сущностям и к интегрированным показателям, полученным за определенный промежуток времени. Уровень ускорения обеспечивает обработку данных в реальном масштабе времени, так как требуемые данные не могут быстро появиться на уровне обслуживания. В работе [8] показано, что существующая схема лямбда-архитектуры имеет ряд существенных недостатков. В частности, если требуется новое пакетное представление, то для его создания необходимо выполнить поиск по всей большой базе данных пакетного уровня, т.е. для каждого нового запроса требуется выполнить поиск в большой базе данных.

В работе [8] была предложена новая блок-схема лямбда-архитектуры (рис. 1).


*Рис. 1*. Новая блок-схема лямбда-архитектуры [8].

Метаданные используются для приближенного вычисления агрегированных значений (sum, avg, count) необходимых атрибутов JSON-документов. Для реализации поискового запроса требуется $n << N$ случайных чтений сегментов базы данных, где $N$ – общее число сегментов, т.е. скорость выполнения запросов существенно увеличивается при незначительной потере точности вычислений агрегированных значений. Уменьшение ошибки оценки этих значений достигается за счет нового метода расчета вероятностей чтения сегментов.

В статье предлагаются решения потоковой обработки данных в системе с лямбда-архитектурой на уровне ускорения.

## Анализ процессов потоковой обработки данных на уровне ускорения

В [9] предлагается целостный подход к организации потоковой обработки данных, который ориентирован, в основном, на реализацию уровня ускорения. Соответствующая архитектурная диаграмма включает следующие звенья:

звено сбора данных;

звено очереди сообщений;

звено анализа;

хранилище данных в памяти;

звено доступа к данным.

Кратко рассмотрим перечисленные звенья потоковой обработки данных на уровне ускорения.

1. *Звено сбора данных*. Здесь используется паттерн «поток». Данные поступают от мобильных устройств (или средств), они предварительно сохраняются в лог-журналах с целью увеличения надежности системы (протоколирование с использованием методов RBML, SBML, HML) и затем передаются на вход следующего звена. Например, данные о выполненных заказах такси непрерывно поступают в систему и там обрабатываются.

2. *Звено очереди сообщений*. В качестве примера можно указать следующие средства обмена сообщениями: NSQ, ZeroMQ, Apache Kafka. Одним из самых популярных решений является проект Apache Kafka [10], отличающийся от аналогов своей надежностью и предоставлением семантики exactly-once [11]. Он позволяет публиковать потоки сообщений и подписываться на них.

В этом звене можно выделить три главных компонента: производитель (звено сбора данных), брокер, потребитель (звено анализа). На рис. 2 приведена схема обмена сообщениями, где буквой «Б» обозначен компонент бро-

кера. От звена сбора данных поступают сообщения, которые брокер ставит в очередь и затем по подписке передает («проталкивает») их брокеру-получателю. Тот ставит сообщения в выходную очередь. Звено анализа посылает запросы брокеру и читает («вытягивает») сообщения из очереди.

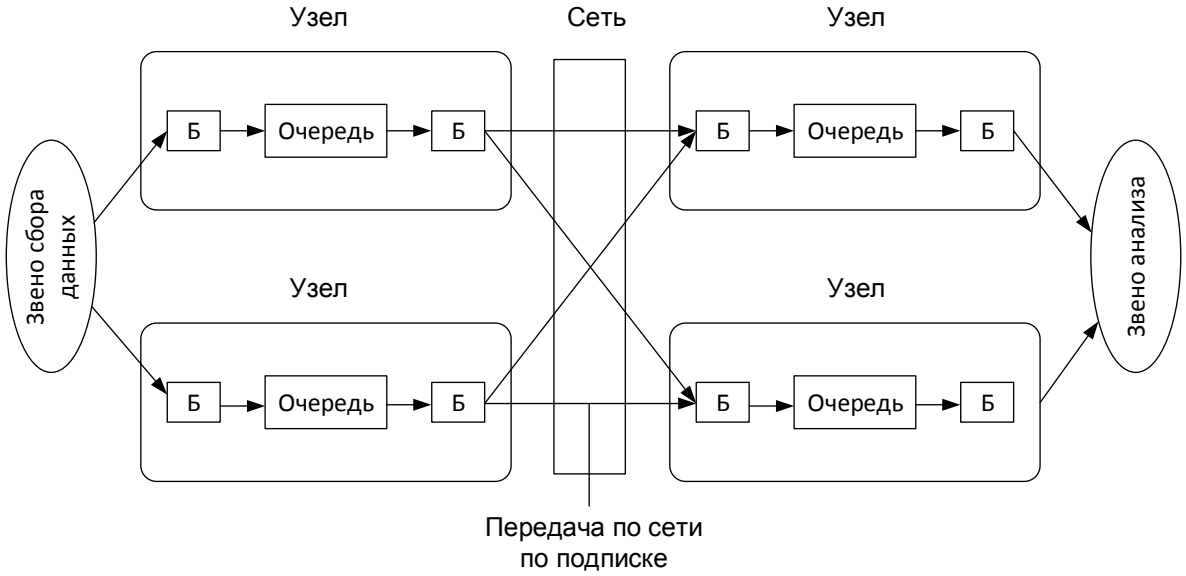


*Рис. 2*. Схема работы брокера.

В зависимости от реализации брокера «проталкивание» может быть заменено на «вытягивание» и наоборот. Брокеры объединены в логический кластер. Параметры звена очереди сообщений должны быть подобраны так, чтобы не было переполнения очередей. Для этого работа брокеров должна быть промоделирована с помощью системы массового обслуживания [12, 13].

После некоторой фильтрации полученные сообщения сохраняются в журнале изменений (см. рис. 1) для их дальнейшей обработки на пакетном уровне и уровне обслуживания (метаданные). Также эти данные поступают в звено анализа уровня ускорения.

3. *Звено анализа*. В настоящее время существует немало технологий анализа данных. Самыми популярными из продуктов с открытым исходным кодом являются Spark Streaming, Storm, Flink и Samza [14 – 16]. Все они – проекты Apache. Перечисленные системы обладают рядом общих черт [9] (рис. 3).

Потоковый диспетчер распределяет приложения анализа (см. ниже) по потоковым процессорам распределенной системы. Сообщения, поступающие из звена очереди сообщений, объединяются в пакеты, которые накапливаются в системе в течение некоторого интервала времени $\Delta$. Далее потоковый диспетчер распределяет пакеты по потоковым процессорам, их обраба-

тывают приложения анализа. Важно, чтобы обработка завершилась за время, меньшее Δ. Потоковый процессор называется по-разному: в Spark Streaming – это «исполнитель Spark», в Storm – «супервизор», в Flink – «исполнитель», в Samza – «исполнитель заданий Samza».

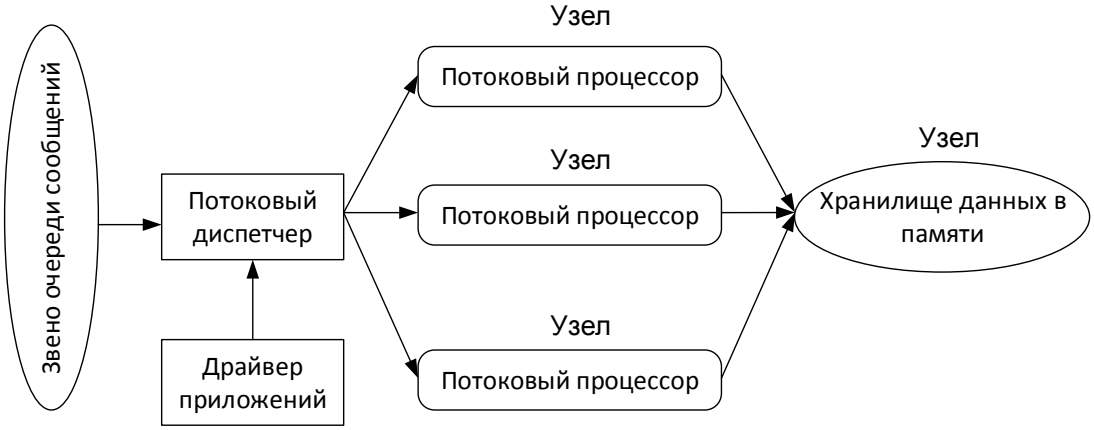


*Рис. 3*. Общая схема звена анализа.

Приложения анализа могут быть различными:

подсчет уникальных значений (на основе битовых комбинаций, например, алгоритмы LogLog, HyperLogLog, HyperLogLog$^{++}$ [17, 18], или на основе порядковых статистик, например, алгоритм MinCount [19]);

подсчет частоты и суммы значений какого-либо элемента в потоке (например, алгоритм Count-Min Sketch [20]);

определение, встречалось ли значение в потоке ранее (алгоритм на основе фильтра Блума [21, 22]);

и другое

4. *Хранилище данных в памяти*. Для поступающих элементов вычисляются хеш-функции, и полученные значения накапливаются (или обновляются) в таблице каждого потокового процесса (рис. 4). Перечисленные в пункте 3 приложения анализа обладают свойством линейности: результирующую таблицу можно получить путем простого сложения (или обновления) локальных таблиц. Вычисление хеш-функций, накопление или обновление таблиц выполняются относительно быстро. Объем каждой таблицы небольшой и составляет несколько килобайтов, поэтому передача их по сети занимает немного времени. Но анализ показывает (см. следующий раздел), что погрешность воспроизведения (по запросу) значений исходных элементов по результирующей таблице может быть достаточно большой.

5. *Звено доступа к данным*. Существует много паттернов взаимодействия потокового клиента (получателя данных) с хранилищем данных [9]: синхронизация данных (Data Sync), удаленный вызов метода или процедуры

(RMI/RPC), простой обмен сообщениями, издатель-подписчик. При этом также необходимо выбрать протокол отправки данных клиентам [9]: веб-уведомления (webhook), длинный HTTP-опрос, протокол пересылаемых сервером событий (Server-Sent Events, SSE), веб-сокеты (WebSocket). Протокол WebSocket (существует с 2011 г.) превосходит по характеристикам остальные протоколы.

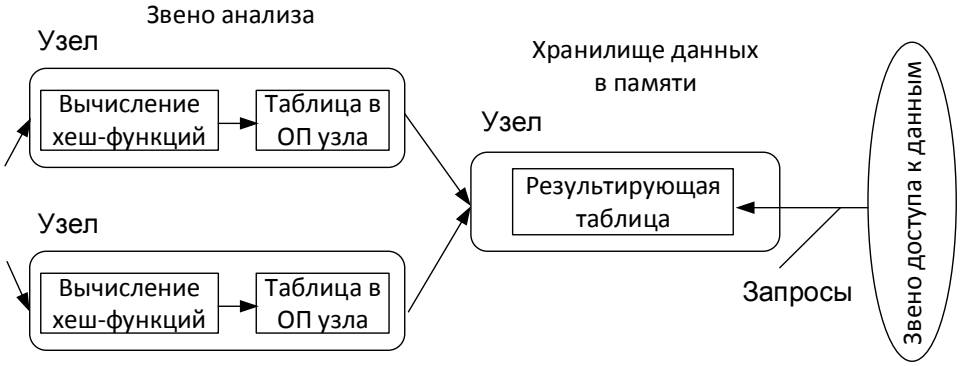


*Рис. 4*. Схема формирования хранилища данных в памяти.

## Алгоритм Count-Min Sketch для подсчета частоты и суммы значений какого-либо элемента в потоке

Подсчет частоты и суммы значений элементов, появляющихся в потоке, является фундаментальной задачей во многих приложениях потоков данных – таких как отслеживание финансовых данных, обнаружение вторжений, мониторинг сети [23], обработка сообщений от мобильных устройств и средств, торговых центров и др.

Для решения этой задачи был разработан алгоритм Count-Min Sketch [20]. Он стал одним из первых для целого класса подобных алгоритмов. В [25] приведена теория распределения эскизов по узлам, учитывающая их свойство линейности. Общая теория эскизов изложена в [26], где также имеются рекомендации по выбору хеш-функций (с. 219).

Рассмотрим алгоритм Count-Min Sketch более подробно [20].

1. *Структура данных.*

Эскиз (sketch) представлен двумерным массивом (таблицей) count[$d$,$w$], где $d$ - число строк, $w$ – число столбцов. Заданы параметры ($\varepsilon$, $\delta$) и $w = \lceil e / \varepsilon \rceil$ и $d = \lceil \ln (1 / \delta) \rceil$, e – основание натурального логарифма. Все элементы массива (ячейки таблицы) первоначально равны нулю. Кроме того, заданы $d$ хэш-функций:

$$h_1 \ldots h_d : \{1 \ldots n\} \rightarrow \{1 \ldots w\}. \quad (1)$$

Будем считать, что $h_k(i)$ – случайная целочисленная величина, которая

равномерно распределена на отрезке [1,$w$] для каждого $i$=1...$n$. Также предполагается, что $\{h_k(i)\}_k$ независимы для каждого $i$. Независимость сохраняется и по $i$.

2. *Обновление эскиза.*

Пусть из потока поступает пара ($i$, $c_i$), где $i$ – номер элемента, $c_i \geq 0$ – его значение (если $c_i$=1, то эскиз используется для подсчета числа появлений этого элемента в потоке, т.е. частоты). К некоторой ячейке каждой строки таблицы добавляется величина $c_i$ (рис. 5). Формально это можно записать так:

$$\text{count}[k, h_k(i)] \leftarrow \text{count}[k, h_k(i)] + c_i,\ k = 1,..., d. \qquad (2)$$

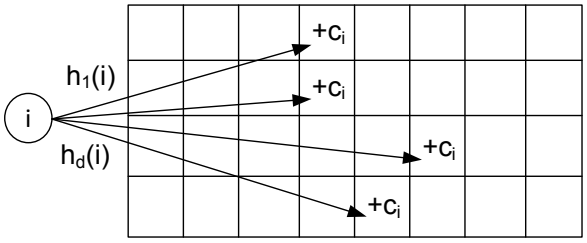


*Рис. 5*. Схема обновления эскиза.

3. *Чтение (восстановление)накопленных значений $c_i$ элемента i ($a_i^*$).*

Восстановленное значение рассчитывается по формуле:

$$a_i^* = \min_k \text{count}[k, h_k(i)]. \qquad (3)$$

4. *Оценка точности восстановленного значения $a_i$*.*

Границы полученной оценки $a_i^*$ имеют следующие значения [20]:

$$a_i \leq a_i^*;\ a_i^* \leq a_i + \varepsilon\|\boldsymbol{a}\|_1 \text{ – с вероятностью не менее } 1 - \delta, \qquad (4)$$

где $a_i$ – точное значение накопления; $\|\boldsymbol{a}\|_1$ – метрика $L_1$.

Для получения правой границы $a_i^*$ было использовано неравенство Маркова. Она может быть большой, все зависит от накопленных величин $L_1 = \Sigma a_i$ (см. (4)). В [24] предлагается уменьшить значение метрики $L_1$ путем вычитания из $\boldsymbol{a}$ некоторого вектора $\boldsymbol{\beta}$ с одинаковыми значениями элементов. Для определения $\boldsymbol{\beta}$ требуется оценить медиану точных величин $\{a_i\}$ по некоторой случайной выборке, которую надо как-то получить. Здесь также для некоторых распределений $\{a_i\}$ правая граница $a_i^*$ может быть большой.

Оценим точность восстановленного значения иначе. Ясно, что если $n \leq w \cdot d$, то использовать эскиз не имеет смысла. В этом случае выгоднее применить вектор длиной $n$: так как и память меньше, и сохраняются точные значения накоплений $\{a_i\}$. Поэтому далее будем полагать, что $n > w \cdot d$.

Оценим сначала вероятность, что восстановленное значение $a_i^*$ не будет совпадать с точным значением $a_i$. Это вероятность, что $\forall k\ \exists(i_1 \neq i)\ (h_k(i_1) = h_k(i))$:

$$p = (1 - (1 - 1/w)^{n-1})^d. \qquad (5)$$

Выражение во внешних скобках соответствует квантору ∃, а степень d – квантору ∀.

Выполним некоторые преобразования (5):

$$p = (1-((1-1/w)^{-w})^{-(n-1)/w})^d \overset{1}{=} (1-e^{-(n-1)/w})^d \overset{2}{=} \\ ((1-1/e^{(n-1)/w})^{-e^{(n-1)/w}})^{-d/e^{(n-1)/w}} \overset{3}{=} e^{-d/e^{(n-1)/w}} \quad (6)$$

Здесь в преобразованиях [1] и [3] был использован 2-й замечательный предел, так как обычно $w \geq 128$ (для 1) и $e^{(n-1)/w} >> 1$ (для 3) в силу $n > w \cdot d$, а $d \geq 8$.

Пусть $n = w \cdot d + 1$ и $d = 8$. Тогда из (6) получим $p = 0{,}997$. Т.е. при $n > w \cdot d$ и при типичных значениях $w \geq 128$ и $d \geq 8$ восстановленное значение $a_i^*$ не будет совпадать с точным значением $a_i$ с вероятностью, почти равной 1. Оценим ошибку восстановления.

В силу свойств хеш-функций (1) накопленные значения $d \cdot \Sigma a_i$ равномерно заполняют ячейки матрицы эскиза (см. рис. 5). На одну ячейку в среднем приходится $(d \cdot \Sigma a_i)/(w \cdot d) = \Sigma a_i / w$ накопленных значений. Следовательно, какое-либо восстановленное значение можно оценить как:

$$a_i^* = \Sigma a_i / w = (n \cdot a^\wedge)/w, \quad (7)$$

где $a^\wedge$ – среднее значение величин $\{a_i\}$.

Относительная погрешность восстановления $a_i$ равна

$$(a_i^* - a_i)/a_i = (n/w) \cdot (a^\wedge / a_i) - 1. \quad (8)$$

Но $n/w > d$, $d$ – число хеш-функций (обычно больше 8). И если $a_i$ не превышает среднего значения, то, как следует из (8), относительная погрешность восстановления может очень большой (сотни процентов).

Итак, можно сделать следующие выводы:

1) если $n \leq w \cdot d$, то использовать эскиз не имеет смысла, в этом случае выгоднее применить вектор (одномерный числовой массив) длиной $n$;

2) если $n > w \cdot d$, то погрешность восстановления накопленных значений $\{a_i\}$ может быть очень большой.

Появились работы, в которых предлагаются методы более сложной обработки потоковых данных, не связанные с эскизами. В публикациях [27, 28] рассматриваются методы, позволяющие обнаружить выбросы в данных (например, большое время подачи машины, много отказов для конкретного водителя, отсутствие машины определенного класса и др.). В работе [29] предлагается модель конвоя (convoy), позволяющая на основе анализа потока данных выявить, например, скопление более q машин на расстоянии меньше esp от каждой за время t. То же можно сделать для скопления людей, которым требуется подать автомобили, и др.

## Предложения по реализации звена анализа в системе с лямбда-архитектурой

В предыдущем разделе было показано, что применение эскиза может привести к большой погрешности восстановления накопленных значений элементов, поступающих из потока. Поэтому вместо эскиза предлагается использовать вектор (одномерный числовой массив) длиной *n*. Сначала создается матрица таких векторов. Каждый вектор матрицы соответствует показателю и ключу или какой-либо комбинации ключей (см. ниже). Также для каждого ключа или комбинации ключей создается хеш-таблица.

Из звена очереди сообщений поступают записи <ключи, показатели>. По ключу или их комбинации из соответствующей хеш-таблицы читается номер *i*, который используется для обновления всех векторов (по смещению *i-1*), соответствующих показателям. Для чтения накопившихся в векторах в течение окна значений (частот, времени и др.) используется select-подобный оператор. Тренды этих значений отображаются на экране, и оператор может выявить в какие-то моменты времени критические ситуации.

Для иллюстрации изложенного подхода рассмотрим предметную область «Обслуженные заказы такси». Из потока поступают записи <ключи, показатели>.

**I.** В качестве показателей (Y) выступают следующие атрибуты (в скобках указаны возможные значения):

1 – заказ обслужен (1 или 0);

2 – время подачи (время подачи машины с момента поступления заказа);

3 – отказ пассажира (1 или 0);

4 – отказ водителя (1 или 0);

5 – машина попала в дорожно-транспортное происшествие (ДТП) (1 или 0);

6 – машина остановлена дорожно-постовой службой (ДПС) (1 или 0);

7 – отрицательный отзыв пассажира (1 или 0).

**II.** В качестве ключей и их комбинаций (X) используются следующие атрибуты:

1 – водитель – ключ;

2 – район подачи (машины) – ключ;

3 – (водитель, район подачи) – комбинация ключей,

4 – класс машины – ключ;

5 – номер машины – ключ;

6 – (водитель, номер машины) – комбинация ключей.

**III.** Имеется несколько хеш-таблиц для ключей и их комбинаций – <ключ, значение>:

1 – <водитель, номер i для всех vector 1.Y>, Y=1..7;

2 – <район подачи, номер i для всех vector 2.Y>, Y=1..7;

3 – <(водитель, район подачи), номер i для всех vector 3.Y>, Y=1..7;

4 – <класс машины, номер i для всех vector 4.Y>, Y=1..7;

5 – <номер машины, номер i для всех vector 5.Y >, Y=1..7;

6 – <(водитель, номер машины), номер i для всех vector 6.Y>, Y=1..7.

**IV.** Обновление векторов vector X.Y (рис. 6).

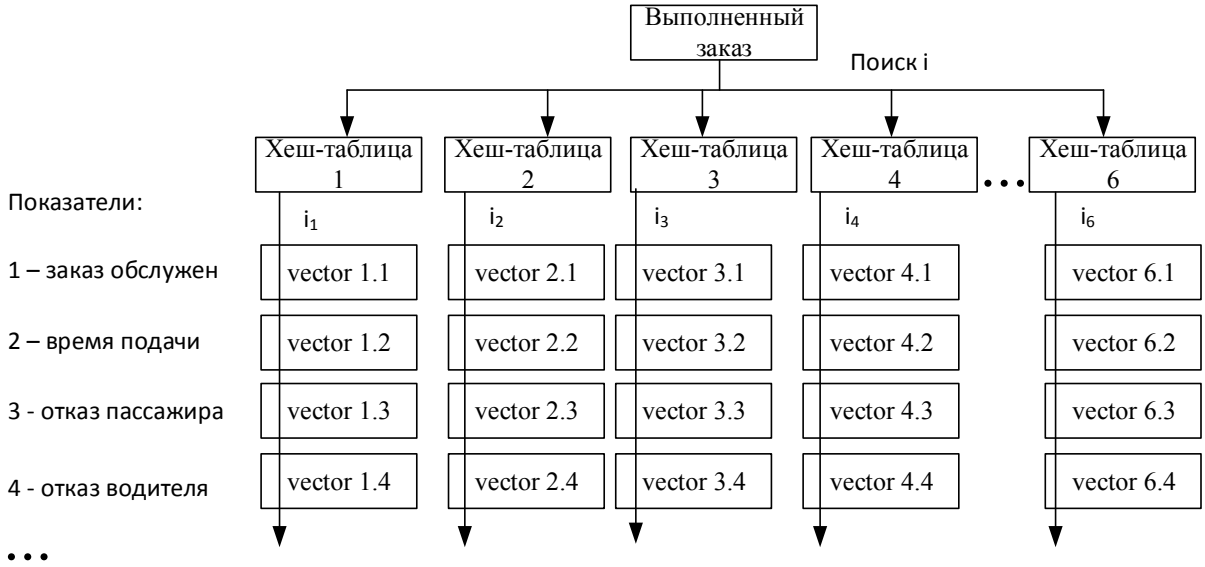


*Рис. 6*. Схема обновления vector X.Y.

В потоке поступает выполненный заказ <ключи, показатели>. Из него выделяются значения ключей: водитель, район подачи, класс машины, номер машины (X = 1, 2, 4, 5). Строятся две комбинации ключей: (водитель, район подачи) и (водитель, номер машины) (X = 3, 6). Из хеш-таблицы X (X = 1...6) по значению ключа хеш-таблицы читается номер $i_X$ для vector X.Y. Номер $i_X$ используется для обновления ячеек (по смещению $i_X - 1$) всех vector X.Y для данного X (Y = 1..7). При этом в зависимости от значения показателя Y к ячейке добавляется (суммируется) либо величина показателя, либо ничего не добавляется (если 0). Если в хеш-таблице X нет соответствующей записи, то она включается и присваивается номер *i* (следующий по порядку), который затем используется для поиска в векторах vector X.Y, Y = 1...7 (номер *i* должен быть уникальным для хеш-таблицы X).

**V.** Примеры запросов.

1. Найти среднее время подачи такси водителем в каком-нибудь районе:

```
select водитель, район подачи, время подачи/заказ обслужен as mean
```

from vector 3.1, vector 3.2

group by водитель, район подачи.

Просматриваются все записи хеш-таблицы 3 (см. рис. 6), для каждого ключа «водитель, район подачи» читается номер $i = i_3$. Этот номер используется для чтения значений «время подачи» (из vector 3.2) и «заказ обслужен» (из vector 3.1). Выполняется деление этих величин.

2. Вывести все показатели для класса машины «Эконом»:

select *

from vector 4.*

where класс машины= «Эконом».

Из хеш-таблицы 4 (класс машины) читается соответствующий номер $i = i_4$. Этот номер используется для чтения значений из vector 4.Y, Y = 1…7.

Запросы выполняются в моменты перемещения окна. Т.е. читаются значения показателей, собранные за интервал времени окна. При этом используется следующий алгоритм:

1) положить T=0, $W=W_1=W_0$ – начальный размер окна (с течением времени может быть плавающим);

2) обнулить все хеш-таблицы и векторы vector X.Y;

3) в момент времени t = T + W окончания окна (размер текущего окна положить равным $W = W_1 = W_0$) или когда номер элемента в потоке больше *n* (размер текущего плавающего окна положить равным W = t – T) активизируется программа, с помощью которой выполняются запросы, отображаются результаты текущего окна, эти значения добавляются к предыдущим результатам для получения трендов:

если $W_1 – W > 0$, то положить $W=W_1=W_1 – W$, // новое плавающее окно

T = t;

4) перейти к пункту 2 алгоритма.

Для доступа к хранилищу данных в памяти можно использовать Web-сокеты (см. звено доступа к данным). После получения от клиента команды «притормози» (клиент перегружен) можно автоматически увеличить размер окна (уменьшить нагрузку $\lambda$ на клиента). Но при этом следует учитывать, что количество номеров элементов в потоке может стать больше размера вектора *n* (см. предыдущий алгоритм).

Использование скользящего окна здесь нецелесообразно, так как в этом случае существенно усложняется ведение хеш-таблиц и векторов на интервале скольжения окна. Потребуется каждый раз выяснять, что нужно удалять в хеш-таблицах и векторах по истечении очередного интервала

скольжения.

Конечно, здесь следует использовать свойство линейности векторов: векторы можно обновлять на разных серверах, а потом в конце окна объединять на координирующем сервере.

Объем векторов, которые хранятся в ОП узла, невелик. Предположим, что $n = w \cdot d = 2^7 \cdot 2^3 = 2^{10}$ (размер одного эскиза). Объем одного вектора равен $v1 = n \cdot 4$ (байтов) = 4КБ. Пусть число хеш-таблиц равно 6 (число ключей и их комбинаций), а число показателей равно 7. Тогда объем всех векторов в ОП узла равен $V = 4(\text{КБ}) \cdot 6 \cdot 7 = 168$ КБ.

Можно выделить следующие преимущества предложенного подхода к реализации звена анализа уровня ускорения:

в динамике можно подключать (или исключать) векторы для новых показателей Y;

в динамике можно подключать (или исключать) хеш-таблицы с новыми ключами или их комбинациями (X);

имеется возможность строить комбинации ключей, что позволяет выполнять запросы select с group by по этим комбинациям;

можно использовать плавающее окно, если число разных элементов в потоке станет больше $n$. Это позволяет экономить память и время обновления вектора, поскольку нет необходимости увеличивать его размер. Размер вектора следует динамически увеличить только в случае перегрузки клиента, выполняющего доступ к хранилищу данных в памяти – и то, если при увеличении размера окна число элементов в потоке становится больше $n$.

## Заключение

Показано, что использование эскиза приводит к большой ошибке восстановления накопленных значений при достаточно большом числе элементов в потоке.

Предложена структура уровня ускорения с использованием векторов для накопления значений элементов, позволяющая в динамике подключать (или исключать) векторы и хеш-таблицы для новых показателей Y и ключей X, поступающих в потоке данных. Имеется возможность в динамике строить комбинации ключей, что позволяет выполнять запросы select к векторам с использованием конструкции group by по этим комбинациям.

Показано, что при переполнении какого-либо вектора возможно уменьшение окна (плавающее окно). Но в этом случае следует учитывать, что увеличивается нагрузка $\lambda$ на клиента, обрабатывающего запросы к хранилищу данных в памяти.

Объем векторов, которые хранятся в ОП узла, невелик. Это позволяет быстро передавать векторы по сети и объединять их на координирующем сервере, используя свойство линейности.

ЛИТЕРАТУРА


1. *Клепплан М.* Высоконагруженные приложения. Программирование, масштабирование, поддержка. – СПб.: Питер, 2018.
2. *Марц Н., Уоррен Д.* Большие данные: принципы и практика построения масштабируемых систем обработки данных в реальном времени. – М.: ООО «И.Д. Вильямс», 2016.
3. *Kiran M. et al.* Lambda architecture for cost-effective batch and speed big data processing // 2015 IEEE International Conference on Big Data (Big Data). IEEE. – 2015. – P. 2785-2792.
4. *Gribaudo M., Iacono M., Kiran M.* A performance modeling framework for lambda architecture based applications // Future Generation Computer Systems. – 2018. – Vol. 86. – P. 1032-1041.
5. *Perrot A. et al.* HeatPipe: High Throughput, Low Latency Big Data Heatmap with Spark Streaming // 2017 21st International Conference Information Visualisation (IV). IEEE. – 2017. – P. 66-71.
6. *Yang F. et al.* The RADStack: Open source lambda architecture for interactive analytics // Proceedings of the 50th Hawaii International Conference on System Sciences. – 2017. – P. 1703-1712.
7. *Spangenberg N., Wilke M., Franczyk B.* A Big Data architecture for intra-surgical remaining time predictions // Procedia computer science. – 2017. – Vol. 113. – P. 310-317.
8. *Григорьев Ю.А., Ермаков О.Ю.* Лямбда-архитектура системы с уровнем обслуживания на основе метаданных для приближенной обработки запросов // Информатика и системы управления. – 2019. – №2. – С. 3-15.
9. *Пселтис Э.Д.* Потоковая обработка данных. Конвейер реального времени. – М.: ДМК Пресс, 2018.
10. *Apache Kafka.* A distributed streaming platform: https://kafka.apache.org/ (дата обращения: 31.03.2020).
11. *Нархид Н., Шапира Г., Палино Т.* Apache Kafka. Потоковая обработка и анализ данных. – СПб.: Питер, 2018.
12. *Wu H., Shang Z., Wolter K.* Performance Prediction for the Apache Kafka Messaging System // 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS). IEEE. – 2019. – P. 154-161.
13. *Kroß J., Krcmar H.* Modeling and simulating apache spark streaming applications //Softwaretechnik-Trends. – 2016. – Vol. 36, №. 4. – P. 1-3.
14. *Quoc D.L. et al.* Approximate stream analytics in apache flink and apache spark streaming // arXiv preprint arXiv:1709.02946. – 2017.
15. *Chintapalli S. et al.* Benchmarking streaming computation engines: Storm, flink and spark streaming // 2016 IEEE international parallel and distributed processing symposium work-

shops (IPDPSW). IEEE. – 2016. – P. 1789-1792.
16. *Noghabi S.A. et al.* Samza: stateful scalable stream processing at LinkedIn //Proceedings of the VLDB Endowment. – 2017. – Vol. 10, №. 12. – P. 1634-1645.
17. *Flajolet P. et al.* Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm // Conference on Analysis of Algorithms. – 2007. – P.127–146.
18. *Heule S., Nunkesser M., Hall A.* HyperLogLog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm // Proceedings of the 16th International Conference on Extending Database Technology. – 2013. – P. 683-692.
19. *Giroire F.* Order statistics and estimating cardinalities of massive data sets // International Conference on Analysis of Algorithms DMTCS proc. AD. – 2005. – Vol. 157. – P. 166.
20. *Cormode G., Muthukrishnan S.* An improved data stream summary: the count-min sketch and its applications // Journal of Algorithms. – 2005. – Vol. 55, №. 1. – P. 58-75.
21. *Bloom B.H.* Space/time trade-offs in hash coding with allowable errors // Communications of the ACM. – 1970. – Vol. 13, № 7. – P. 422-426.
22. *Tarkoma S., Rothenberg C.E., Lagerspetz E.* Theory and practice of bloom filters for distributed systems. // IEEE Communications Surveys and Tutorials. –2012. – № 14(1). – P.131-155.
23. *Basat R. B., Friedman R., Shahout R.* Stream frequency over interval queries // Proceedings of the VLDB Endowment. – 2018. – Vol. 12, №. 4. – P. 433-445.
24. *Chen J., Zhang Q.* Bias-Aware Sketches // Proceedings of the VLDB Endowment. – 2017. – Vol. 10, № 9. – P. 961-972.
25. *Cormode G., Garofalakis M.* Sketching streams through the net: Distributed approximate query tracking // Proceedings of the 31st international conference on Very large data bases. – 2005. – P. 13-24.
26. *Cormode G. et al.* Synopses for massive data: Samples, histograms, wavelets, sketches // Foundations and Trends® in Databases. – 2011. – Vol. 4. – № 1–3. – C. 1-294.
27. *Yoon S., Lee J.G., Lee B.S.* NETS: extremely fast outlier detection from a data stream via set-based processing // Proceedings of the VLDB Endowment. – 2019. – Vol. 12, №. 11. – P. 1303-1315.
28. *Cao L. et al.* Efficient discovery of sequence outlier patterns // Proceedings of the VLDB Endowment. – 2019. – Vol. 12, №. 8. – C. 920-932.
29. *Orakzai F., Calders T., Pedersen T.B.* k/2-hop: fast mining of convoy patterns with effective pruning // Proceedings of the VLDB Endowment. – 2019. – Vol. 12, №. 9. – P. 948-960.

*E-mail:*
*Григорьев Юрий Александрович – grigorev@bmstu.ru;*
*Ермаков Олег Юрьевич – ihelos.ermakov@gmail.com.*