# Statshouse

Система сбора статистики и мониторинга ВКонтакте. Старая версия системы также известна, как “Статлоги”.

Это АЛЬФА-версия документа от 6 января 2022.

# Какие задачи мы решаем

1. Сбор статистики с высоким временным разрешением (1 секунда) и маленькой задержкой (10 секунд), почти real-time графики.
2. Нормальная работа системы при неожиданном резком росте обьёма вставляемых данных, в случае ошибок в коде клиентов или каких-то событий.
3. Защита одних пользователей от других и честность распределения общих ресурсов, для избежания трагедии общин в такой “полудикой” среде, как ВК.
4. Инфраструктура мониторинга максимально отделена от инфраструктуры, которую мы мониторим. В идеале Statshouse должен продолжать работать, если недоступно всё, кроме него.
5. Простота интеграции (создания клиентских библиотек), минимально число зависимостей.
6. Ограничение кардинальности ключей, эффективность хранения и получения данных.
7. Потенциальную возможность создания в будущем open source продукта, для привлечения сторонних ресурсов к развитию Statshouse.

# Архитектура системы

## Компоненты

- Источник данных (демон statshouse) - принимает статистику по UDP в форматах JSON, ProtoBuf, MessagePack, TL а также по протоколу RPC TL, валидирует и интерпретирует статистику, накапливает её в течение секунды и отправляет в аггрегаторы по протоколу RPC TL. В случае недоступности аггрегаторов, хранит данные на локальном диске в пределах квоты и отправляет позже. В VK источников около 15000.
- Аггрегатор статистики - принимают данные от источников, аггрегирует статистику, соответствующую каждой секунде от всех источников и вставляет в

базу данных. Аггрегаторов столько, сколько шардов-реплик clickhouse используется, в VK 5 шардов по 3 реплики, то есть 15 аггрегаторов.
- База данных (clickhouse) - хранит статистику
- Сервис доступ к данным (демон statshouse-api) - позволяет делать только эффективные запросы к базе данных с помощью достаточно узкого API, кэширует данные для минимизации нагрузки на базу данных. Мы максимально ограничиваем выборку данных напрямую из clickhouse, так как неэффективные запросы могут негативно повлиять на кластер.
- UI - получает данные от statshouse-api и показывает статистику пользователям.
- Ingress proxy - принимает данные от источников извне DC и направляет на аггрегаторы.
- Хранилище настроек метрик (MySQL) - хранит список метрик и настройки для каждой метрики.
- Сервис глобального словаря (PMC) - хранит отображения строковых меток в целые и обратно.

# Модель данных

## Метрики и аггрегация

Единицей сбора, просмотра и настройки свойств статистики является метрика. При записи метрики каждое измерение дополняется набором меток (ключей, тэгов), а также временем (timestamp). Измерения с одинаковыми наборами меток аггрегируются, как в пределах временного интервала, так и между машинами.

Предположим, что в гипотетическом продукте нам нужно знать количество принятых пакетов в секунду, причём пакеты могут иметь разный формат, а также быть корректными и некорретными, соответственно для каждого пакета мы хотим знать статус обработки - “ок”, либо одна из нескольких ошибок. Каждый раз, когда пользовательский код принимает пакет, он отправляет на вход системы (в statshouse) событие, например в формате JSON.

```
{"metrics":[ {"name": "toy_packets_count",
 "tags":{"format": "JSON", "status": "ok"},
 "counter": 1}] }
```

Формат и статус могут иметь несколько разных значений каждый

```
{"metrics":[ {"name": "toy_packets_count",
 "tags":{"format": "TL", "status": "error_too_short"},
 "counter": 1} ]}
```

Если представить событие в виде строчки в традиционной базе данных, то после аггрегации в пределах секунды у нас может получиться что-то вроде такой таблички - для каждой комбинации меток, которая встретилась, нам понадобится строчка, чтобы подсчитать число событий с данной комбинацией меток.

| timestamp | metric | format | status | counter |
|---|---|---|---|---|
| 13:45:05 | toy_packets_count | JSON | ok | 100 |
| 13:45:05 | toy_packets_count | TL | ok | 200 |
| 13:45:05 | toy_packets_count | TL | error_too_short | 5 |

Количество необходимых строчек в такой табличке называется кардинальностью метрики, для этой секунды на этой машине кардинальность метрики 3. Обьём данных не зависит от того, сколько было самих событий, а зависит только от того, сколько было разных наборов меток. Строчки с одинаковым набором меток “схлопываются”, а счётчики суммируются между собой.

Сбор данных происходит одновременно на многих машинах-источниках. После того, как сбор и аггрегация данных за секунду завершены, данные отправляются на машины-аггрегаторы, которые аггрегируют данные со всех источников. Для нашей гипотетической метрики после агрегации между машинами для каждой секунды у нас получится что-то вроде

| timestamp | metric | format | status | counter |
|---|---|---|---|---|
| 13:45:05 | toy_packets_count | JSON | ok | 1100 |
| 13:45:05 | toy_packets_count | JSON | error_too_short | 40 |
| 13:45:05 | toy_packets_count | JSON | error_too_long | 20 |
| 13:45:05 | toy_packets_count | TL | ok | 30 |
| 13:45:05 | toy_packets_count | TL | error_too_short | 2400 |
| 13:45:05 | toy_packets_count | msgpack | ok | 1 |

После аггрегации между машинами кардинальность может увеличиться, так как каждая отдельная машина не всегда встречает все возможные комбинации меток, в

данном случае общая кардинальность для данной секунды 6.

После аггрегации данные записываются а базу данных в посекундную табличку. Обьём посекундных данных велик, поэтому происходит постоянная ротация и посекундные данные доступны за 2 суток.

Параллельно у каждой строчки зануляются секунды и данные аггрегируются в пределах минуты и записываются в минутную табличку. Аналогично в часовую. Поминутные данные хранятся месяц, почасовые не удаляются (или удаляются вручную).

Общая часовая кардинальность метрики очень важна, так как именно она определяет, сколько рядов для данной метрики будет храниться длительное время в базе данных.

Также при выборке информации нам придётся пробежать все строчки за выбранный интервал времени, именно кардинальность определяет, сколько строк нам придётся в среднем пробежать и насколько быстро это удастся сделать.

## Сэмплирование

У системы есть 2 узких места, первое - отправка данных с машин-источников на машины-аггрегаторы и обработка ими, второе - вставка аггрегаторами в базу данных. Если каждую секунду будет отправляться больше данных, чем может обработать аггрегатор или вставить база данных, они начнут копиться, и появится отставание, которое в прицнипе может расти бесконечно, а может и исчезнуть если обьём данных уменьшится. Чтобы гарантировать минимальное отставание, применяется сэмлирование. Все “лишние” данные будут выброшены, а оставшиеся домножены на коэфициент таким образом, чтобы сохранить мат ожидания значений.

Предположим, что за секунду собраны 3 строки данных, причём все принадлежат одной метрике

| timestamp | metric | format | status | counter |
|---|---|---|---|---|
| 13:45:05 | toy_packets_count | JSON | ok | 100 |
| 13:45:05 | toy_packets_count | TL | ok | 200 |
| 13:45:05 | toy_packets_count | TL | error_too_short | 5 |

А ширина канала позволяет отправить на аггрегатор только 2 ряда. Тогда будет выбран sampling factor 1.5, ряды случайно перемешаны, и отправлены только первые 2 ряда со счётчиком домноженным на 1.5. Таким образом могут быть отправлены следующие данные.

| timestamp | metric | format | status | counter |
|---|---|---|---|---|
| 13:45:05 | toy_packets_count | TL | ok | 300 |
| 13:45:05 | toy_packets_count | JSON | ok | 150 |

Или такие

| timestamp | metric | format | status | counter |
|---|---|---|---|---|
| 13:45:05 | toy_packets_count | TL | ok | 300 |
| 13:45:05 | toy_packets_count | TL | error_too_short | 7.5 |

Или такие

| timestamp | metric | format | status | counter |
|---|---|---|---|---|
| 13:45:05 | toy_packets_count | TL | error_too_short | 7.5 |
| 13:45:05 | toy_packets_count | JSON | ok | 150 |

Отметим, что нецелые факторы сэмплирования приводят к тому, что значения счётчиков перестают быть целыми. Поэтому в системе statshouse счётчики не ограничены целыми значениями, а изначально имеют тип с плавающей точкой.

Хотя на каждой машине каждую секунду часть рядов будет выбрасываться, после аггрегации между машинами или между секундами в среднем мы получим примерно те же данные, как и без сэмплирования, лишь добавится шум.

Чем выше кардинальность метрики - тем выше будут выбраны коэффициенты сэмплирования, и тем сильнее будет шум на графике. Это стимулирует пользователей уменьшать кардинальность.

Одинаковый алгоритм применяется как перед отправкой источниками на аггрегатор, так и перед вставкой аггрегатором в базу данных.

Рассмотрим теперь, что происходит, если у нас более 1 метрики и ширину канала нужно распределить между ними.

## Честное сэмплирование

Мы хотим, чтобы метрики минимально влияли друг на друга. Если одна метрики внезапно стала генерировать гораздо больше рядов, чем другая, желательно, чтобы именно для этой метрики был выбран больший коээфициент сэмплирования, чтобы остальные метрики не были затронуты.

Алгоритм работает таким образом:

- Сначала все метрики сортируются по возрастанию количества занятых рядов, и берутся по-очереди.
- Для каждой метрики
  - Вычисляется положенное ей число рядов - оставшийся бюджет делится на количество оставшихся метрик
  - Если метрика занимает меньше положенного, данные записываются целиком без сэмплирования.
  - Если же метрика занимает больше положенного, она сэмплируется таким образом, чтобы занять положеное количество рядов (если положено 5, а фактическое 20, то будет выбран sampling factor 4).
  - Оставшийся бюджет уменьшается на число рядов, использованных метрикой.

Так как размер в байтах каждого ряда зависит от типа метрики и количества использованных меток, алгоритм на самом деле работает не с рядами, а с байтами.

Также, поскольку в реальности есть более важные и менее важные метрики, предусмотрено задание вручную веса для каждой метрики. Метрике с весом 2 будет выделен канал в 2 раза более широкий, чем метрике с весом 1.

Сэмплирование также является важным стимулом для каждого пользователя системы не использовать больше, чем справедливый процент ресурсов системы, так как при росте обьёма записи выше справедливого происходит ухудшение качества статистики.

## Выбор временного разрешения

Для некоторых метрик секундное разрешение не так важно, как минимальный коэффициент сэмплирования. Поэтому Statshouse позволяет установить для каждой метрике меньшее разрешение, например 10 секунд. Тогда данные будут отправляться в 10 раз реже, а число рядов, выделяемой метрике окажется примерно в 10 раз больше. При этом задержка увеличится примерно на 20 секунд - сначала 10 секунд данные будут собираться, а затем ещё 10 секунд отправляться, каждую секунду по 1/10 собранных данных. Такой способ отправки гарантирует

честное распределение канала между метриками с разным временным разрешением. Разрешение должно быть делителем числа 60.

## Хранение меток - имена

Все метрики хранятся в одной таблице, где есть 15 колонок для меток, названных key1..key15. Когда в примере выше мы использовали метки с именем “format”, “status” на самом деле система выбирала одну из колонок на основании описания метрики, например можно отредактировать описание так, чтобы format направлялся в key1, а статус - в key2 или наоборот. Также можно использовать системные имена key1..key15 напрямую и они имеют приоритет над заданными пользователем, так что при изменении имён меток можно временно использовать системные имена.

Дополнительная колонка key0 имеет специальную интерпретацию, и служит для задания окружения (environment), в котором собирается статистика. Например production или staging. В принципе, могут использоваться любые значения. Например, если на подмножестве машин экспериментальная версия, может быть задана строчка для этого эксперимента. Задаётся в клиентских библиотеках один раз при инициализации. В остальном с точки зрения остальной системы key0 ничем не отличается от остальных колонок.

## Хранение меток - значения

Так как значения меток постоянно повторяются, для компактности хранения и скорости записи и выборки key0..key15 имеют тип int32 и там хранятся не строчки, а значения, которые берутся из гигантского отображения строчек string ↔ int32. Отображение общее на все метрики, а его элементы никогда не удаляются, а чтобы предотвратить его бесконтрольный рост, на создание элементов отображения установлен суточный бюджет в 10 значений для каждой метрики с лимитом в 500. (TODO - пока вместо бюджета исторически используется простой лимит в 100 значений в сутки). Когда бюджет исчерпывается и отображение создать не удаётся, в колонку записывается специальное служебное значение, чтобы не терять событие целиком.

Максимальная длина строчки 128 байтов, если больше - обрезается. Строчке с обеих сторон делается TRIM, а непечатные символы заменяются на дорожный знак.

Иногда значения меток уже имеют подходящий тип, а количество значений велико, например номера приложений или каких-нибудь других обьектов. Тогда можно отредактировать описание метрики, указав, что определённые ключи являются “сырыми”, тогда для них вместо отображения строчки-значения, она будет просто парситься, как int32 и вставляться в таблицу как есть.

## Хранение времени и окно приёма

Обычно время события назначается системой автоматически по времени приёма события, однако если нужно писать старую статистику, можно указать конкретное значение, но принимается только статистика за последние полтора часа, если более старая - время будет установлено в текущее время минус полтора часа. Это сделано потому, что система может работать только при эффективной аггрегация между хостами, а для этого все хосты должны присылать данные за конкретную секунду максимально вместе и слажено. А также система хранения (сейчас это clickhouse) работает гораздо медленнее, когда в неё вставляют старые данные. Для тех метрик, которые явно указывают время событий можно ожидать выбора системой более высоких факторов сэмплирования, так как ряды с разным временем не могут быть аггрегированы между собой. TODO - потестировать, насколько увеличение окна приёма ухудшает работу clickhouse, и, если возможно, расширить окно.

## Метрики-значения

Кроме метрик-счётчиков, есть метрики-значения. Например вместо счётчика принятых пакетов мы бы могли захотеть записывать размер принятых пакетов.

```
{"metrics":[ {"name": "toy_packets_size",
 "tags":{"format": "JSON", "status": "ok"},
 "value": [150]} ]}
```

или

```
{"metrics":[ {"name": "toy_packets_size",
 "tags":{"format": "JSON", "status": "error_too_short"},
 "value": [0]} ]}
```

Значение является массивом, чтобы можно было отправлять сразу несколько значений пачкой, что более эффективно.

Тогда кроме счётчика будут вычисляться аггрегаты значений - сумма, минимальное и максимальное значение.

| timestamp | metric | format | status | counter | sum | min | max |
|---|---|---|---|---|---|---|---|
| 13:45:05 | toy_packets_size | JSON | ok | 100 | 13000 | 20 | 1200 |
| 13:45:05 | toy_packets_size | TL | ok | 200 | 7000 | 4 | 800 |
| 13:45:05 | toy_packets_size | TL | error_too_short | 5 | 10 | 0 | 8 |

Среднее вычисляется при выборке данных путём деления суммы на сумму счётчика.

## Перцентили значений

Если в описании метрики установлена галочка “с перцентилями”, то кроме аггрегатов значений система будет считать перцентили на источниках, пересылать на аггрегаторы, аггрегировать там и записывать в clickhouse. Объём данных для такой метрики значительно возрастёт, поэтому система скорее всего выберет высокие факторы сэмплирования. В таком случае может помочь уменьшение кардинальности или временного разрешения. TODO - достаточно хорошие более компактные перцентили.

## Метрики-счётчики уникумов

Используются, чтобы оценить число разных уникальных целых значений (если значения не целые, а например строки, то можно взять hash строк)
Предположим, что пакеты содержат id пользователя, отправляющего пакеты. Мы можем посчитать сколько разных пользователей отправляло пакеты.

```
{"metrics":[ {"name": "toy_packets_user",
 "tags":{"format": "JSON", "status": "ok"},
 "unique": [17]} ]}
```

Уникальное значение является массивом, чтобы можно было отправлять сразу несколько значений пачкой, что более эффективно.

Множества хранятся в сжатом виде и использованием функции, подобной Hyper Log Log, так что сами значения недоступны, можно узнать только оценку кадинальности множеств.

| timestamp | metric | format | status | counter | unique |
|---|---|---|---|---|---|
| 13:45:05 | toy_packets_user | JSON | ok | 100 | uniq(17, 19, 13, 15) |
| 13:45:05 | toy_packets_user | TL | ok | 200 | uniq(17, 19, 13, 15, 11) |
| 13:45:05 | toy_packets_user | TL | error_too_short | 5 | uniq(51) |

## Топ строк

Иногда бывает ситуация, когда число разных значений метки огромно и неограниченно, например referrer или слово в поисковом запросе. Если использовать обычные метки, то новые значения очень быстро выберут бюджет на создание элементов отображения, да ещё и засорят его огромным количеством одноразовых значений. Для подобных случаев система поддерживает специальную метку с именем skey, что означает буквально строковый ключ. Для каждой комбинации обычных ключей  на источнике создаётся специальная структура данных, которая хранит некоторое количество популярных за эту секунду значений skey (например, 100), когда структура наполняется, применяется вероятностное вытеснение. На аггрегатор отправляется топ этих значений, например 10. При сэмплировании выбрасываются либо все строки из набора, либо ни одной, таким образом распределение самих наборов по пространству кардинальности сохраняется. Аггрегатор собирает все строки для набора, и в свою очередь вставляет в базу данных топ этих строк (например, 20) плюс специальную пустую строку как маркер значения “остальные”.

Пока что такие метрики работают только, как счётчики. В дальнейшем позволим им работать и как метрики-значения, и метрики-счётчики уникумов (TODO).

## Метка машины-источника и механизм max_host

Большинство пользователей интуитивно хотят использовать имя машины-источника в качестве метки, чтобы иметь возможность просматривать статистику с каждой

машины независимо. Однако, что может оказаться неожиданным, добавление такой метки перкращает аггрегацию данных между машинами, а значит увеличивает кардинальности и обьём данных в соответствующее число раз. Например, если машин-источников 100, то обьём данных увеличится в 100 раз, и системой могут быть выбраны гигантские коэффициенты сэмплирования, например 10, 50 или 100, при этом качество данных сильно ухудшается из-за шума.

Поэтому Statshouse использует для всех метрик (TODO - пока только для счётчиков и значений) механизм, когда в специальную колонку max_host при аггрегации данных записывается имя машины, ответственной за максимальное значение, либо внёсшей максимальный вклад в счётчик. Такая колонка увеличивает обьём данных менее, чем на 10%, при этом позволяет ответить на вопросы “на каком хосте максимальный обьём занятого дискового пространства” или “на каком хосте максимальное количество ошибок каждого типа”.

Прежде чем добавлять метку с именем машины-источника мы рекомендуем попробовать посмотреть в UI функцию max_host для ваше метрики. Чаще всего, зная имя лишь одной проблемной машины, можно посмотреть логи и решить проблему.

Также советуем, где возможно, использовать разбивку не по отдельным машинам, а по их группам, используя метку environment или собственную метку. Например, запуская экспериментальную версию на одной или нескольких машинах, можно установить environment staging или dev для того, чтобы отделить статистику с этих машин от остальных.

## Метаметрики

Для получения сведений о работе самой системы на разных этапах собираются и записываются метаметрики. Все они имеют префикс “__builtin”. Самые главные из них вынесены в UI для каждой метрики - это ошибки приёма данных (например отрицательное значение счётчика или значение-NaN), факторы сэмплирования, выбранные источником и аггрегатором, а также оценка часовой кардинальности метрики и количество созданых элементов отображения.

Многие метаметрики по разным причинам не подчиняются общим правилам, например не сэмплируются. Некоторые метаметрики, например статус приёма данных и факторы сэмлирования источником пересылаются в специальной компактной форме для экономии трафика.

# Детали реализации

## Метрики и метаинформация

Свойства каждой метрики хранятся в специальной таблице описания метрик в MySQL, для каждой метрики хранится её тип (для правильного отображения), имя метрики, имена и способ интерпретации ключей.

Метрики создаются через UI, автоматического создания метрик не происходит. Это важно, так как все компоненты предполагают, что разных метрик немного (максимум десятки тысяч) и не имеют защиты от неконтролируемого роста числа разных метрик.

Аггрегаторы формируют из таблицы MySQL логический журнал обновления, используя триггер автоматически устанавливающий update_time с индекс по этой колонке, а источники постоянно находятся в TL RPC long poll на изменение журнала, поэтому изменения свойств метрики уже через несколько секунд отражаются на всех источниках.

Удалять метрики нельзя, так как нет способа сделать это эффективно в базе данных Clickhouse. Поэтому используется скрытие метрики установкой флага visible (это действие обратимо). При этом статистика по этой метрике перестаёт записываться в базу данных.

TODO - не решена проблема того, что ненужные метрики продолжают писаться, так как нет стимула их удаления.

TODO - перенести таблицу описания метрик в clickhouse на шарде 1.

TODO - запретить прямой доступ

## Отображение строковых значений

Отображение хранится в key-value базе данных PMC.

Хранится прямое и обратное отображение

```
stlogs_s_iphone => 12
stlogs_i_12 => ‘iphone’
```

А также счётчик последнего созданного отображения.

```
stlogs_key_id_gen2 => 123 456 789
```

При создании каждый аггрегатор атомарно увеличивает счётчик, затем использует функцию “создать или взять если существует” для создания прямого отображения с этим значением счётчика, если результат функции совпал, значит именно этот аггргеатор создал значение, и тогда он создаёт обратное отображение.

Если при создании обратного отображения произошла ошибка, может нарушиться консистентность. TODO - создавать обратное отображение всегда, когда читают прямое.

Flood-лимиты на создание хранятся в той же базе данных.

```
stlogs_krestr_metric => 20, TTL = 86400
```

Перед созданием отображения сначала атомарно увеличивается лимит, если превышен то отображение не создаётся. Когда проходит TTL, лимит сбрасывается.

Все остальные ветки в PMC не используются современной версией системы, и могут быть удалены.

TODO - реализовать медленно восполняемые бюджеты с помощью функции replaceIfEqual

Аггрегаторы работают с базой данных напрямую, и кэшируют отображения в памяти и файлах (на месяц), источники работают с отображениями через аггрегаторы, и также кэшируют отображения в памяти и файлах (тоже на месяц). Также источники в коде имеют boostrap из 100000 самых распространённых отображений, это нужно так как при разворачивании на 10000 машинах пришлось бы скачать с аггрегаторов примерно миллиард значений, что заняло бы долгое время.

TODO - перенести в open source key-value базу данных, развернуть на тех же стойках, где стоит clickhouse.

TODO - запретить прямой доступ

## Приём данных по TL RPC

```
statshouse.metric#3325d884 fields_mask:#
  name:    string
  tags:    %(Dictionary string)
  counter: fields_mask.0?%double
  t:       fields_mask.5?%long // UNIX nanoseconds UTC
  value:   fields_mask.1?%(Vector double)
  unique:  fields_mask.2?%(Vector long)
= statshouse.Metric;

@write statshouse.addMetricsBatch#56580239 fields_mask:#
  metrics:%(Vector %statshouse.Metric)
= True;
```

Ошибки приёма возвращаются, как TL ошибки. Мы не специфицируем коды ошибок, так как не предполагаем никакой логики в клиентах при получении ошибок, только запись в лог и последующий анализ вручную. Все ошибки также пишутся во встроенную метрику __builtin_ingestion_status.

Счётчики и значения имеют тип float64, и при приёме им делается clamp в диапазон значений [-max(float32..max(float32)]. Это сделано для того, чтобы при их суммировании и других операциях над ними, в том числе внутри базы данных, никогда не получать значений +-inf.

Значения-уникумы имеют тип int64, который интерпретируется просто как 64 бита и выбран вместо uint64 только потому, что в некоторых языках и TL нет типа uint64. В любом случае, эти значения считаются чем-то вроде хэшей и с ними не делается никаких операций, кроме вычисления кардинальности составленных из них множеств.

## Приём данных по UDP

Формат TL, совпадающий с сериализацией функции addMetricBatch. Также форматы JSON, Protobuf, MessagePack со структурой идентичной TL.

Пакет может содержать более 1 экземпляра addMetricBatch, следующих один за другим без какой-либо дополнительной разметки. Однако все экземпляры внутри пакета должны быть в одном формате. Например для JSON:

```
{"metrics":[
{"name":"rpc_call_latency",
 "tags":{"protocol": "tcp"},
 "value": [15, 18, 60]},
{"name": "rpc_call_errors",
 "tags":{"protocol": "udp","error_code": "-3000"},
 "counter": 5}
]}
{"metrics":[
{"name": "external_landings",
 "tags":{"country": "ru","sex": "m","skey": "lenta.ru"},
 "counter": 1}
]}
```

## Приём данных по unix datagram socket или TCP (TODO)

Если можно делать неблокирующую отправку в unix datagram socket, то можно использовать их вместо UDP для того, чтобы отследить потерю пакетов со стороны отправителя.

Либо клиенты, например PHP, могут с той же целью пользоваться non-blocking TCP/unix socket для того, чтобы не блокироваться, когда буфер сокета переполняется (хвостик пакета, не влезший целиком запоминается, и будет отправлен по мере освобождения буфера. Новые же пакеты будут выбрасываться целиком, а их счётчик будет увеличиваться).

## Структура основных таблиц Clickhouse

Вся статистика всех типов для всех метрик исходно сохраняется в одну таблицу clickhouse, один ряд соответствует одному аггрегату. Примерное определение таблицы такое.

```
CREATE TABLE statshouse2_value_1s (
    `time`           DateTime,
    `metric`         Int32,
```

```
    `key0`           Int32,
    `key1`           Int32,
...
    `key15`          Int32,
    `skey`           String,
    `count`          SimpleAggregateFunction(sum, Float64),
    `min`            SimpleAggregateFunction(min, Float64),
    `max`            SimpleAggregateFunction(max, Float64),
    `sum`            SimpleAggregateFunction(sum, Float64),
    `max_host`       AggregateFunction(argMax, String, Int64),
    `percentiles`    AggregateFunction(quantilesTDigest(0.5), Flo
at32),
    `uniq_state`     AggregateFunction(uniq, Int64)
) ENGINE = *MergeTree
PARTITION BY toDate(time) ORDER BY (metric, time, key0, key1,
..., key15, skey);
```

Если метрик не является перцентилем или счётчиком уникумов, то значение в соответствующей колонке будет пустым. Также, для обычного счётчика все колонки, кроме count будут нулевыми/пустыми. Аналогично колонка skey будет непустой только если использован топ строк.

Данные из этой таблицы аггрегируются за некоторые временные интервалы (например, 60 секунд, 3600 секунд) и сохраняются в идентичные таблицы, но с другим именем, для поддержки быстрой выборки за временные интервалы, значительно превышающие секунду.

TODO - пока что не решена задача ограничения часовой кардинальности, то есть обеспечения того, что в часовой таблице для каждой метрики будет не более фиксированного количества рядов в час. Использование детерминистского сэмплирования по hash(hour, metric, tags), когда каждый час в таблицу пишется подмножество меток решает задачу, но на практике такая статистика превращается в мусор и не представляет никакой ценности. Также малопрактичным является удаление меток, пока кардинальность не уменьшится до приемлемой. В любом случае, метрики с высокой часовой кардинальностью в основном вредят только себе, так как при выборке данных по метрике важен только обьём данных, занимаемой ей, а не остальными метриками.

Данные шардируются между шардами clickhouse по хэшу от (metric, key0, ... , key15), поэтому части данных метрики, соответствующие, например, метке “protocol”:”tcp” и “protocol”:”udp” могут оказаться на разных шардах, и для получения полной картины нужно всегда делать Distributed Query ко всем шардам. Это так, потому что набор ключей имеет ограниченную кардинальность, и при её достижении обьём данных при аггрегации перестаёт расти, поэтому если бы каждый шард хранил “сэмпл” всей статистики, то при достаточнм обьёме данных, фактически каждому шарду пришлось бы хранить число рядов равное кардинальности статистики, а не долю, пропорциональную числу шардов.

Буферные таблицы не используются, так как обычно каждый аггрегатор делает 1 вставку в секунду. Вставка делается в incoming-таблицу, а копирование оттуда с помощью материализованного представления с фильтрацией значений по time в пределах окна приёма данных (двое суток), это защита от ошибочной вставки мусорных данных, которая приведёт к тому, что clickhouse будет пережимать прошлые данные, а это может сделать кластер недоступным на недели и месяцы.

Количество реплик каждого шарда должно быть ровно 3. Количество шардов может быть каким угодно. Для предотвращения неправильной конфигурации источников и разного шардирования, которое бы привело к взрывному росту обьёма данных из-за плохой аггрегации, источники присылают номер шарда-реплики с каждым пакетом данных а аггрегатор, и если аггрегатор видит, что данные предназначаются не ему, отвечает ошибкой. Этот же номер шарда-реплики используется ingress proxy для того, чтобы направить данные на правильный аггрегатор.

## Ingress Proxy

Поскольку источники и аггрегаторы используют протокол TL RPC с ключом шифрования датацентра, источники вне датацентра не могут напрямую присоединяться к аггрегаторам, так как это бы потребовало копирования/раскрытия ключа датацентра на внешние площадки.

Поэтому ingress proxy имеет отдельный набор ключей шифрования для подключения извне. Любой ключ может отзываться простым удалением из конфигурации ingress proxy.

Ingress proxy не имеет состояния, и для уменьшения вероятности атаки проксирует только подмножество типов запросов TL RPC, используемых аггрегаторами.

## Детали приёма регулярных значений

Источники финализирую секунду для отправки синхронно с календарной секундой. Поэтому если у клиента есть какое-то значение, которое должно фиксироваться каждую секунду (условный “уровень воды”), клиентские библиотеки стараются присылать это значение в районе середины календарной секунды. Таким образом увеличивается вероятность того, что каждая секунда будет содержать ровно 1 измерений, однако гарантий этого система не предоставляет.

## Взаимодействие между источником и аггрегатором, детали

Аггрегатор имеет 2 логических точки входа для данных, одна для отправки “актуальных” данных, другая - для отправки “исторических” данных, то есть тех, которые не удалось отправить сразу после создания.

Аггерагтор всегда приоритизирует вставку актуальных данных, поэтому после сбоя, когда сначала данные долго не могли вставиться, но потом нормальный ход вставки возобновился, актуальные данные начинают вставляться немедленно, а вот накопившийся за время сбоя обьём исторических данных будет вставлен по возможности, настолько быстро, насколько это не мешает вставке актуальных данных.

Аггрегатор позволяет вставлять актуальные данные за последние 5 секунд (короткое окно, настраивается), если источник не успел, ему отправляется ответ “присылай данные, как исторические”. Для каждой актуальной секунды аггрегатор хранит контейнер со статистикой, куда и аггрегируется данные клиентов (TODO - тут картинка), как только наступает следующая секунда, данные секунды, выходящей из короткого окна вставляются в clickhouse, а источники получают ответ с результатом вставки. Также короткое окно распространяется на две секунды в будущее, чтобы нормально работали клиенты, у которых часы немного идут вперёд.

Аггрегатор позволяет вставлять исторические данные за последние 2 суток (длинное окно, настраивается). Если приходят более старые данные, пишется специальная метастатистика, данные выбрасываются, а на источник отправляется ответ OK.

Поскольку аггрегация данных между машинами очень важна, каждый источник делает запрос на вставку  нескольких десятки исторических секунд, начиная от самой старой, аггрегатор принимает все эти запросы, затем выбирает самую старую секунду, аггрегирует, вставляет и присылает ответ, затем снова выбирает самую старую секунду, и так далее. Такой алгоритм приводит к тому, что отставшие сильнее источники “догоняют” менее отставших, таким образом создаётся тенденция аггрегировать и вставлять каждую историческую секунду максимально одновременно, один раз.

Поскольку аггрегатор должен лимитировать обьём вставленных в секунду данных, а часть данных может вставляться позже, как исторические, аггрегатор учитывает, сколько источников внесли вклад в секунду, и устанавливает лимит пропорционально (все источники каждую секунду присылают метаданные, даже если никаких пользовательских событий не было). Поэтому, если актуальные данные за какую-то секунду прислали 80% источников, то для их вставки будет использовано 80% канала, затем если исторические данные прислали ещё 15% источников, для их вставки будет выделено ещё 15% канала, и т.д. Этот алгоритм приводил к слишком большому сэмплированию, когда источников очень мало, поэтому в этом случае к лимиту делается небольшая поправка в сторону увеличения.

Каждый шард-реплика источника при старте выбирает случайную поправку к часам от -1 до 0 секунд, для того, чтобы источники не присылали данные лавиной в момент переключения секунды, приводя к большому количеству потерянных пакетов.

Источник сохраняет данные на диск в случае получения ошибки от аггрегатора или его недоступности.

## Предотвращение двойной вставки и работа при отказе аггрегатора

Если источник получает от аггрегатора ошибку, и после этого отправляет те же самые данные другому аггрегатору, находящемуся на другой машине, то для дедупликации нужна система консенсуса. Мы решили, что сложность такой системы слишком высока, поэтому у нас возможна ситуация, когда и основной и запасной аггрегатор вставят данные от одного источника за одну секунду. Мы решили, что это происходит нечасто, и вместо того, чтобы предотвратить двойную вставку, мы контролируем её специальной метаметрикой “количество источников, приславших данные в эту секунду”. Для того, чтобы эта метаметрика была

стабильной при нормальной работе, источники присылают данные каждую секунду, даже если никакой пользовательской статистики в эту секунду не было. Также, если источник обнаруживает, что часы сдвинулись вперёд больше, чем на секунду (например, машина затупила или уснула), он присылает разницу в специальном поле, чтобы аггрегаторы могли учесть это в специальной метаметрике.

При отказе аггрегатора, данные, предназначеныые ему, отправляются источниками на один из двух аггрегаторов-реплик, распределяясь между ними по номеру секунды - чётные секунды идут на одну из оставшихся, нечётные на другую, так что в среднем нагрузка на каждый вырастает на 50%. Это одна из причин того, что мы поддерживаем только 3 реплики clickhouse. TODO - убрать ограничение в 3 реплики.

## Система мониторинга и новая система watchdogs - TODO

Есть желание, чтобы настройка watchdogs максимально совпадала с настройками для графиков.

Каждый Watchdog будет следить за тем, чтобы некоторое значение сильно не увеличивалось и не уменьшалось (с настройкой чувствительности). Значения будут автоматически “исправляться” с учётом дневных и недельных колебаний (TODO - отключаемо?). Было бы здорово, чтобы Watchdog не только включал тревогу, когда значение вышло из рамок, но и отключал тревогу, когда значение вернётся обратно в норму.

Возможно, Watchdogs это отдельная группа машин, каждая их которых пишет собственную статистику, и поэтому реагирует на отказы соседей, тогда они будут иметь свой отдельный clickhouse, куда аггрегаторы будут записывать данные, аггрегированные по тем ключам, по которым нет white/black листов ни в одном Watchdog.

Возможно, Watchdogs будут работать без обращения к базе clickhouse, на основании данных, собираемых аггрегатором. Так их работа не будет зависеть от latency вставки и выборки из clickhouse, которая может быть довольно непредсказуема.