
PICODATA

*Распределенный сервер приложений со встроенной распределенной базой данных*

# Руководство пользователя

# Оглавление

# О данном руководстве

Документ «Руководство пользователя» содержит сведения, которые должны помочь пользователям и системным администраторам запускать программное обеспечение Picodata и использовать его в своей работе.

Информация об инсталляции программного обеспечения приведена в отдельном документе «Руководство по установке».

Информация о внутреннем устройстве распределенной системы (кластера) приведена в отдельном документе «Руководство администратора».

В текущем документе содержится описание параметров запуска и последовательности действий, необходимой для развертывания и поддержания работоспособности распределенного кластера СУБД.

Сведения в данном документе относятся к текущей публично доступной версии ПО Picodata 22.11.0, вышедшей в ноябре 2022 г. Информация в этом руководстве будет обновляться для наиболее полного соответствия фактической функциональности ПО Picodata на момент публикации.

Дата публикации: 12.12.2022

# Общее описание продукта

Данный раздел содержит общие сведения о продукте Picodata, его назначении, области применения и внутреннем устройстве.

## Что такое Picodata?

Программное обеспечение Picodata — это распределенная система промышленного уровня для управления базами данных, а также среда выполнения приложений. Исходный код Picodata открыт. Программное обеспечение Picodata реализует хранение структурированных и неструктурированных данных, транзакционное управление данными, языки запросов SQL и GraphQL, а также среду выполнения приложений (хранимых процедур) на языках программирования Rust и Lua.

## Назначение

Основным назначением продукта Picodata является горизонтально масштабируемое хранение структурированных и неструктурированных данных, управление ими, предоставление среды вычислений внутри кластера, состоящего из реплицированных отдельных узлов (*инстансов*). Данная комбинация возможностей позволяет эффективно реализовать сценарии управления наиболее востребованными, часто изменяющимися, *горячими* данными. В традиционных корпоративных архитектурах для ускорения и повышения надёжности доступа к данным классических, универсальных СУБД используются кэши и шины данных. Использование ПО Picodata позволяет заменить три компонента корпоративной архитектуры — кэш, шина и витрина доступа к данным — единым, высокопроизводительным и строго консистентным решением.

## Задачи

Программное обеспечение Picodata решает следующие задачи:

- реализация общего линеаризованного хранилища конфигурации, схемы данных и топологии кластера, встроенного в распределенную систему управления базами данных;
- предоставление графического интерфейса и интерфейса командной строки по управлению топологией кластера;
- реализация runtime-библиотек по работе с сетью, файловому вводу-выводу, реализация кооперативной многозадачности и управления потоками, работа со встроенной СУБД средствами языка Rust;
- поддержка языка SQL для работы как с данными отдельного инстанса, так и с данными всего кластера;
- управление кластером;
- поддержка жизненного цикла приложения в кластере, включая версионирование, управление зависимостями, упаковку дистрибутива, развертывание и обновление запущенных приложений.

## Область применения

Кластер Picodata обеспечивает быстрый доступ к данным внутри распределенного хранилища. Это позволяет использовать его в следующих областях:

- управление телекоммуникационным оборудованием;
- банковские и в целом финансовые услуги, биржевые торги, аукционы;
- формирование персональных маркетинговых предложений с привязкой ко времени и месту;
- обработка больших объемов данных в реальном времени для систем класса “интернет вещей” (IoT);
- игровые рейтинговые таблицы;
- и многое другое!

## Особенности кластера Picodata

Кластер с СУБД Picodata обладает следующими свойствами:

- автоматическое горизонтальное масштабирование кластера;
- более простая настройка для запуска шардированного кластера. Требуется меньше файлов конфигурации;
- совместимость с любыми инструментами развертывания инстансов (Ansible, Chef, Puppet и др.);
- обеспечение высокой доступности данных без необходимости в кластере Etcd и дополнительных настройках;
- автоматическое определение активного инстанса в репликасетах любого размера;
- единая схема данных во всех репликасетах кластера;
- возможность обновлять схему данных и менять топологию работающего кластера, например добавлять новые инстансы. Picodata автоматически управляет версиями схемы;
- встроенные инструменты для создания и запуска приложений.

# Архитектура кластера

## Составные части кластера

Архитектура кластера Picodata предполагает систему отдельных *инстансов* — программных узлов, входящих в состав кластера. Каждый такой узел может выполнять различные роли, например роль хранения данных, роль сервера приложения, или служебную роль координатора кластера. Все инстансы работают с единой схемой данных и кодом приложения. Каждый процесс базы данных выполняется на одном процессорном ядре и хранит используемый набор данных в оперативной памяти. Любой отдельный инстанс является частью набора реплик, который также называют *репликасетом*. Репликасет может состоять из одного или нескольких инстансов — дубликатов одного и того же набора данных. Внутри репликасета всегда есть *активный* инстанс и — если реплик больше 1 — то некоторое число *резервных* инстансов, обеспечивающих отказоустойчивость системы в случае выхода из строя или недоступности активного инстанса. Число реплик определяется *фактором репликации*, заданным в глобальных настройках Picodata.

На рисунке ниже показана схема простого кластера из двух репликасетов, каждый из которых состоит из двух инстансов (активного и резервного):

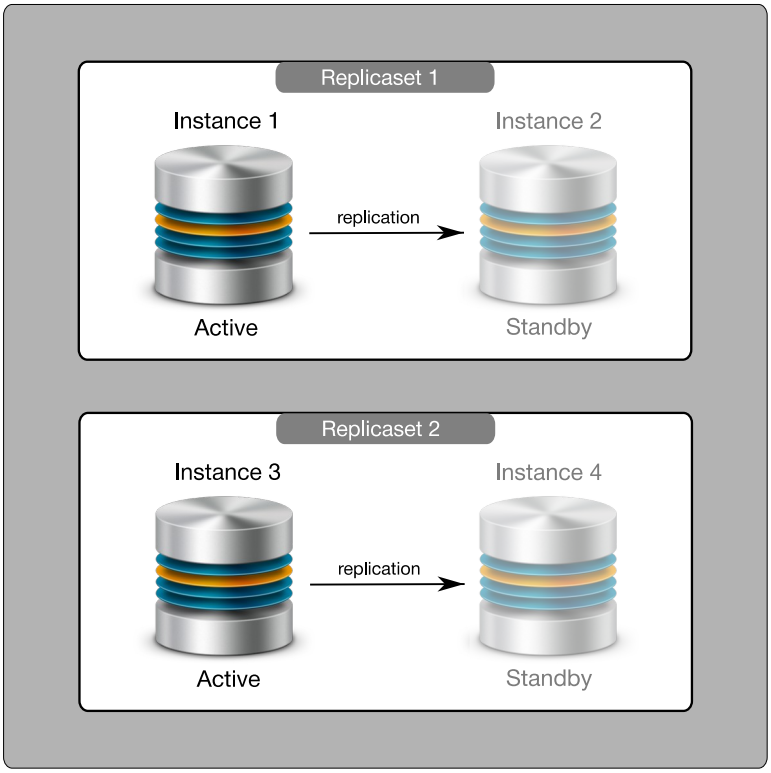


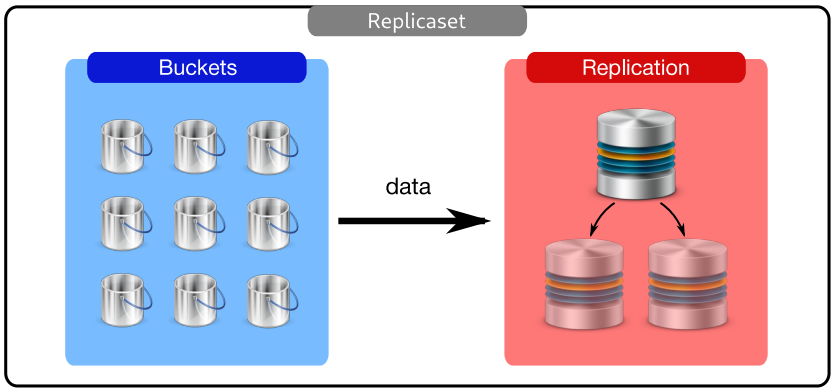

*Схема кластера*

Репликасеты являются единицами горизонтального масштабирования кластера. Данные балансируются между ними автоматически.

## Хранение данных

Внутри каждого репликасета есть *бакет* (bucket) — виртуализированная неделимая единица хранения, обеспечивающая локальность данных (например, хранение нескольких связанных с клиентом записей на одном физическом узле сети). Сам по себе бакет не имеет ограничений по емкости и может содержать любой объем данных. Горизонтальное масштабирование позволяет распределить бакеты по разным шардам, оптимизируя производительность кластера путем добавления новых реплицированных инстансов. Чем больше репликасетов входит в состав кластера, тем меньше нагрузка на каждый из них. Бакет хранится физически на одном репликасете и является промежуточным звеном между данными и устройством хранения. В каждом репликасете может быть много бакетов (или не быть ни одного). Внутри бакета данные задублированы по всем инстансам в рамках репликасета в соответствии с фактором репликации. Количество бакетов может быть задано при первоначальной настройке кластера. По умолчанию кластер Picodata использует 3000 бакетов.

На схеме ниже показан пример схемы хранения данных внутри репликасета:

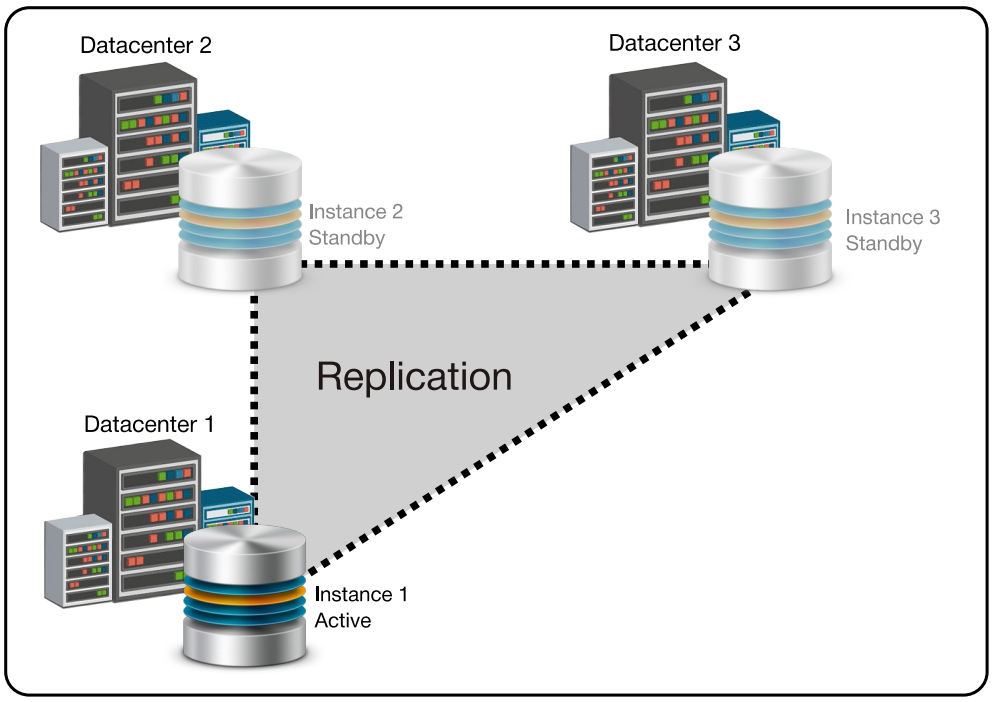



*Хранение данных*

## Отказоустойчивость

Наличие нескольких реплик внутри репликасета обеспечивают его отказоустойчивость. Дополнительно для повышения надежности каждый инстанс кластера внутри репликасета находится на разных физических серверах, а в некоторых случаях — в удаленных друг от друга датацентрах. Таким образом, в случае недоступности датацентра в репликасете происходит переключение на резервную реплику/инстанс без прерывания работы.

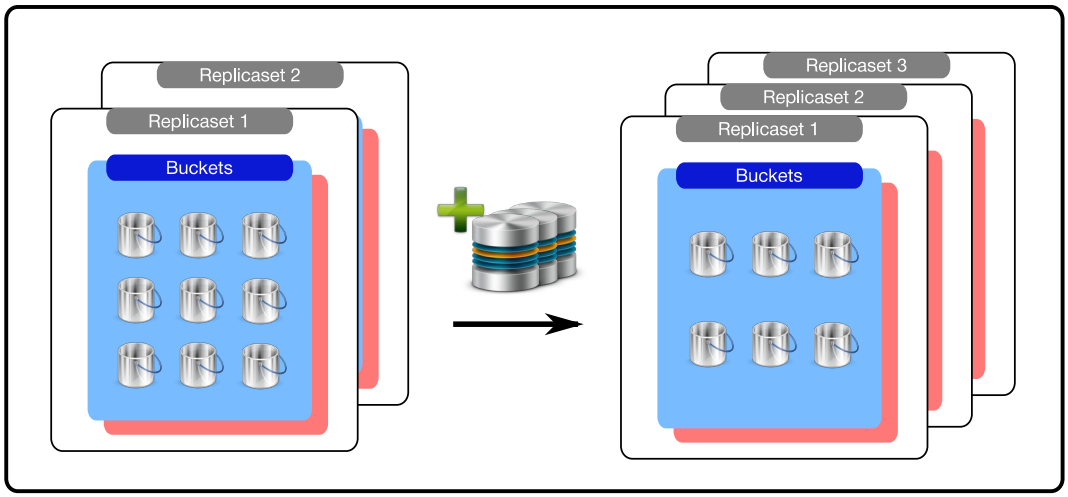
Пример географического распределения репликасета показан на схеме ниже:




*Отказоустойчивость*

## Шардирование

Шардирование — это распределение бакетов между различными репликасетами. В Picodata используется основанное на хэшах шардирование с хранением данных в виртуальных бакетах. Каждый репликасет является *шардом*, и чем больше репликасетов имеется в кластере, тем эффективнее данная функция может разделить массив данных на отдельные наборы данных меньшего размера. При добавлении новых инстансов в кластер и/или формировании новых репликасетов Picodata автоматически равномерно распределит бакеты с учетом новой конфигурации. Пример автоматического шардирования при добавлении в кластер новых инстансов показан на схеме ниже:




*Шардирование*

Таким образом, каждый инстанс (экземпляр Picodata) является *частью репликасета*, а каждый репликасет — *шардом*, а шарды распределены между несколькими серверами.

## Общая схема инициализации кластера

Данный раздел содержит описание архитектуры Picodata, в том числе высокоуровневый процесс инициализации кластера на основе нескольких отдельно запущенных экземпляров Picodata (инстансов).

Администратор запускает несколько инстансов, передавая в качестве аргументов необходимые параметры:

```
picodata run --instance-id i1 --listen i1 --peer i1,i2,i3
picodata run --instance-id i2 --listen i2 --peer i1,i2,i3
picodata run --instance-id i3 --listen i3 --peer i1,i2,i3
# ...
picodata run --instance-id iN --listen iN --peer i1
```

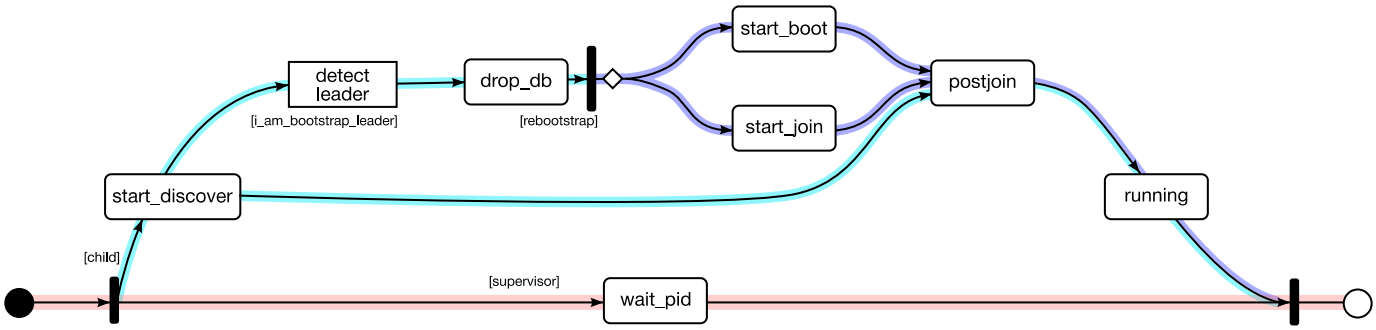
Независимо от количества запускаемых инстансов, в опции `--peer` у каждого из них следует указать один и тот же набор из нескольких инстансов — одного обычно достаточно, но для подстраховки можно взять три. Именно на их основе будет произведена инициализация кластера и поиск всех работающих инстансов для их включения в состав кластера (discovery).

Подробности алгоритма discovery приведены в отдельном документе. В контексте сборки кластера важно лишь понимать, что этот алгоритм позволяет не более чем одному инстансу (peer’у) создать Raft-группу, т.е. стать инстансом с raft_id=1. Если таких инстансов будет несколько, то и Raft-групп, а следовательно и кластеров Picodata получится несколько.

Топологией Raft-группы управляет алгоритм Raft, реализованный в виде крейта `raft-rs`.

### Этапы инициализации кластера

На схеме ниже показаны этапы жизненного цикла инстанса в контексте его присоединения к кластеру Picodata.




*Жизненный цикл инстанса*

Красным показан родительский процесс, который запущен на всем протяжении жизненного цикла инстанса. Вся логика, начиная с присоединения к кластеру, и заканчивая обслуживанием клиентских запросов, происходит в дочернем процессе (голубой цвет). Единственное предназначение родительского процесса — иметь возможность сбросить состояние дочернего (выполнить rebootstrap) и инициализировать его повторно (сиреневый цвет).

Данная схема наиболее полно отражает логику кода в файле `main.rs`. Ниже описаны детали выполнения каждого этапа и соответствующей программной функции.

### *fn main()*

На этом этапе происходит ветвление (форк) процесса `picodata`. Родительский процесс (supervisor) ожидает от дочернего процесса сообщения по механизму IPC и при необходимости перезапускает дочерний процесс. Также, при необходимости дочерний процесс может попросить родителя удалить все файлы БД, т.е. вызвать функцию `drop_db()`. Это может понадобиться для повторной инициализации кластера когда, например, у инстанса изначально имеется временный, случайно сгенерированный `replicaset_id`.

### *fn start_discover()*

Дочерний процесс начинает свое существование с запуска модуля `box.cfg()` и вызова функции `start_discover()`. Возможно, что при этом из БД будет ясно, что bootstrap данного инстанса уже был произведен ранее и что Raft уже знает о вхождении этого инстанса в кластер — в таком случае никакого discovery не будет, инстанс сразу перейдет к этапу `postjoin()`. В противном случае, если место инстанса в кластере еще не известно, алгоритм discovery определяет значение флага `i_am_bootstrap_leader` и адрес лидера Raft-группы. Далее инстанс сбрасывает свое состояние (этап rebootstrap), чтобы повторно провести инициализацию `box.cfg()`, теперь уже с известными параметрами. Сам лидер (единственный с `i_am_bootstrap_leader == true`) выполняет функцию `start_boot()`. Остальные инстансы переходят к функции `start_join()`.

### *fn start_boot()*

В функции `start_boot` происходит инициализация Raft-группы — лидер генерирует и сохраняет в БД первые записи в журнале. Эти записи описывают добавление первого инстанса в пустую Raft-группу и создание начальной clusterwide-конфигурации. Таким образом достигается однообразие кода, обрабатывающего эти записи.

Сам Raft-узел на данном этапе еще не создается. Это произойдет позже, на стадии `postjoin()`.

### *fn start_join()*

Вызову функции `start_join()` всегда предшествует rebootstrap (удаление БД и перезапуск процесса), поэтому на данном этапе в БД нет ни модуля box, ни пространства хранения. Функция `start_join()` имеет простое устройство:

Инстанс-клиент отправляет запрос `raft_join` лидеру Raft-группы (он известен после discovery). После достижения консенсуса в Raft-группе лидер присылает в ответе необходимую информацию: - Идентификатор `raft_id` и данные таблицы `raft_group` — для инициализации Raft-узла; - Идентификаторы `instance_uuid`, `replicaset_uuid` и параметры `replication`, `read_only` для `box.cfg`.

Получив все настройки, инстанс использует их в `box.cfg()`, и затем создает в БД группу `raft_group` с актуальными адресами других инстансов. Без этого инстанс не сможет отвечать на сообщения от других членов Raft-группы. Для того чтобы записи в `raft_group` не были заменены на менее актуальные из журнала Raft, каждая запись маркируется значением `commit_index`.

По завершении этих манипуляций инстанс также переходит к этапу `postjoin()`.

### *fn postjoin()*

Логика функции `postjoin()` одинакова для всех инстансов. К этому моменту для инстанса уже инициализированы корректные пространства хранения в БД и могут быть накоплены записи в журнале Raft. Инстанс инициализирует узел Raft, который начинает взаимодействовать с Raft-группой. В случае, если других кандидатов нет, инстанс тут же избирает себя лидером группы.

В этом месте также устанавливается `on_shutdown` callback, который обеспечит корректное завершение работы инстанса. Следующим шагом инстанс оповещает кластер о том, что он готов проходить настройку необходимых подсистем (репликации, шардинга, и т.д.). Для этого лидеру отправляется запрос на обновление `target_grade` текущего инстанса до уровня `Online`, после чего за дальнейшие действия будет отвечать специальный поток управления topology governor, также называемый `governor_loop`.

Как только запись с обновленным грейдом будет зафиксирована в Raft, узел готов к использованию.

## Обработка запросов

### *#[proc] fn raft_join()*

Значительная часть всей логики по управлению топологией находится в хранимой процедуре `raft_join`. Аргументом для нее является следующая структура:

```
struct join::Request {
    cluster_id: String,
    instance_id: Option<String>,
    replicaset_id: Option<String>,
    advertise_address: String,
    failure_domain: FailureDomain,
}
```

Ответом служит структура:

```
struct JoinResponse {
    /// Добавленный пир (чтобы знать все ID)
    instance: Instance,
    /// Голосующие узлы (чтобы добавляемый инстанс мог наладить контакт)
    peer_addresses: Vec<PeerAddress>,
    /// Настройки репликации (чтобы инициализировать репликацию)
    box_replication: Vec<String>,
}

struct Instance {
    // всевозможные идентификаторы
```

```
    raft_id: RaftId,
    instance_id: String,
    instance_uuid: String,
    replicaset_id: String,
    replicaset_uuid: String,

    // текущее местоположение, виртуальное и физическое
    peer_address: String,
    failure_domain: FailureDomain,

    // целевая и текущая оценки уровня инстанса с точки зрения лидера кластера
    target_grade: CurrentGrade,
    current_grade: CurrentGrade,

    /// Индекс записи в Raft-журнале. Препятствует затиранию
    /// более старыми записями по мере применения Raft-журнала.
    commit_index: RaftIndex,
}
```

Цель такого запроса сводится к добавлению нового инстанса в Raft-группу. Для этого алгоритма справедливы следующие тезисы:

- `join::Request` отправляет всегда неинициализированный инстанс.
- В зависимости от того, содержится ли в запросе `instance_id`, проводится анализ его корректности (уникальности).
- В процессе обработки запроса в Raft-журнал добавляется запись `op::PersistPeer { peer }`, которая помимо всевозможных ID содержит поля `current_grade: Offline`, `target_grade: Offline`, играющие важную роль в обеспечении надежности кластера (подробнее о них в разделе **Topology governor**).
- В ответ выдается всегда новый `raft_id`, никому другому ранее не принадлежавший.
- Генерировать значение `raft_id` может только лидер Raft-группы.
- Помимо всевозможных ID, ответ содержит список голосующих членов Raft-группы. Они понадобятся новому инстансу для того чтобы знать адреса соседей и общаться с ними.
- Также ответ содержит параметр `box_replication`, который требуется для правильной настройки репликации.

## Обработка записей Raft-журнала

Последовательность состояний каждой отдельной записи в Raft-журнале можно описать так:

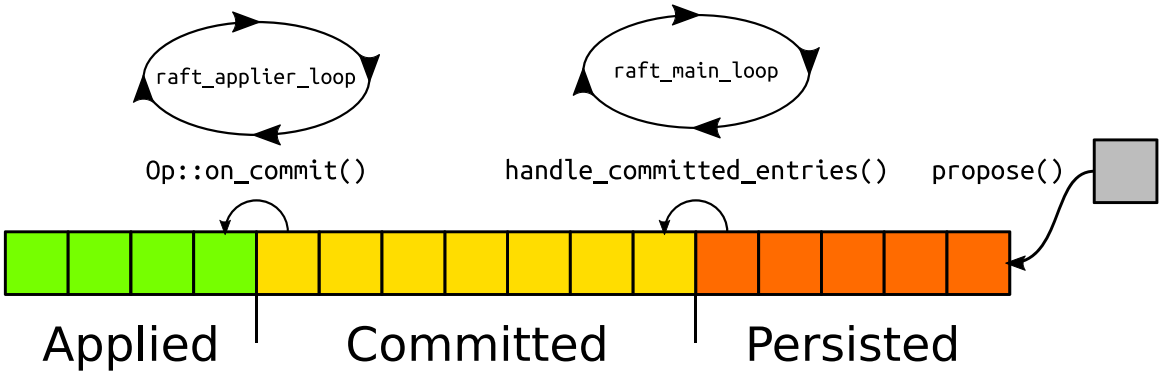
`Persisted` → `Committed` → `Applied`

При добавлении в журнал (по правилам это делает лидер) запись получает статус `Persisted` и начинает реплицироваться (это асинхронно делает файбер `raft_main_loop` ). Когда кворум узлов подтверждает персистентность записи, она считается зафиксированной. Важно понимать, что статус `Committed` присваивается записи на основе совокупности полученной информации, а не какого-то конкретного действия.

Конкретные действия по обработке той или иной записи выполняет отдельный поток `raft_applier`. Для каждой записи он выполняет обработчик `Op::on_commit()` и по завершении присваивает записи статус `Applied`. Важно помнить, что обновление статуса и сама операция могут выполняться не атомарно (если в `Op::on_commit()` происходит

передача управления другому потоку — yield). В таком случае, следует позаботиться хотя бы об идемпотентности операции.

Схема ниже иллюстрирует эту информацию.




*Raft log*

Стоит также помнить, что алгоритм Raft гарантирует лишь консистентность последовательности записей, но не конкретные сроки выполнения. Смена статусов на разных инстансах так или иначе происходит в разные моменты времени, и иногда эту очередность приходится учитывать в алгоритмах. Например, при корректном запланированном выводе инстанса из строя (graceful shutdown) может возникнуть ситуация, когда инстанс завершится слишком быстро, и его соседи могут ошибочно продолжать считать его голосующим. Причиной такой ситуации может быть критерий остановки, включающий ожидание коммита лишь локально на завершаемом инстансе, но не на других — это может стать причиной потери кворума в кластере.

## Graceful shutdown

Чтобы выключение прошло штатно и не имело негативных последствий, необходимо следить за соблюдением следующих условий:

- Инстанс не должен оставаться голосующим, пока есть другие кандидаты в состоянии `Online`.
- Инстанс не должен оставаться лидером.

Чтобы этого добиться, каждый инстанс при срабатывании триггера `on_shutdown` отправляет лидеру запрос `UpdatePeerRequest { target_grade: Offline }`, обработкой которого займется вышеупомянутый `governor_loop`. После этого инстанс пытается дождаться применения записи о смене своего `current_grade` на `Offline` (о том, почему так произойдет см. ниже).

## Описание уровней (grades) кластера

По некоторым причинам коммит записи может не успеть дойти до инстанса в срок, отведенный на выполнение триггера `on_shutdown` триггера (например в кластере может быть потерян кворум). В таком случае корректное завершение работы инстанса (`graceful shutdown`) невозможно.

## Topology governor

В отличие от других кластерных решений (например, того же Tarantool Cartridge) Picodata не использует понятие “состояния” для описания отдельных инстансов. Вместо этого теперь применяется новое понятие «грейд» (grade). Данный термин отражает не состояние самого инстанса, а конфигурацию остальных участников кластера по отношению к нему. Существуют две разновидности грейдов: текущий (`current_grade`) и целевой (`target_grade`). Инициировать изменение `current_grade` может только лидер при поддержке кворума, что гарантирует консистентность принятого решения (и поддерживает доверие к системе в плане отказоустойчивости).

Инициировать изменение `target_grade` может кто угодно — это может быть сам инстанс (при его добавлении), или администратор кластера командой `picodata expel` либо нажатием Ctrl+C на клавиатуре. `target_grade` — это желаемое состояние инстанса, в которое тот должен прийти.

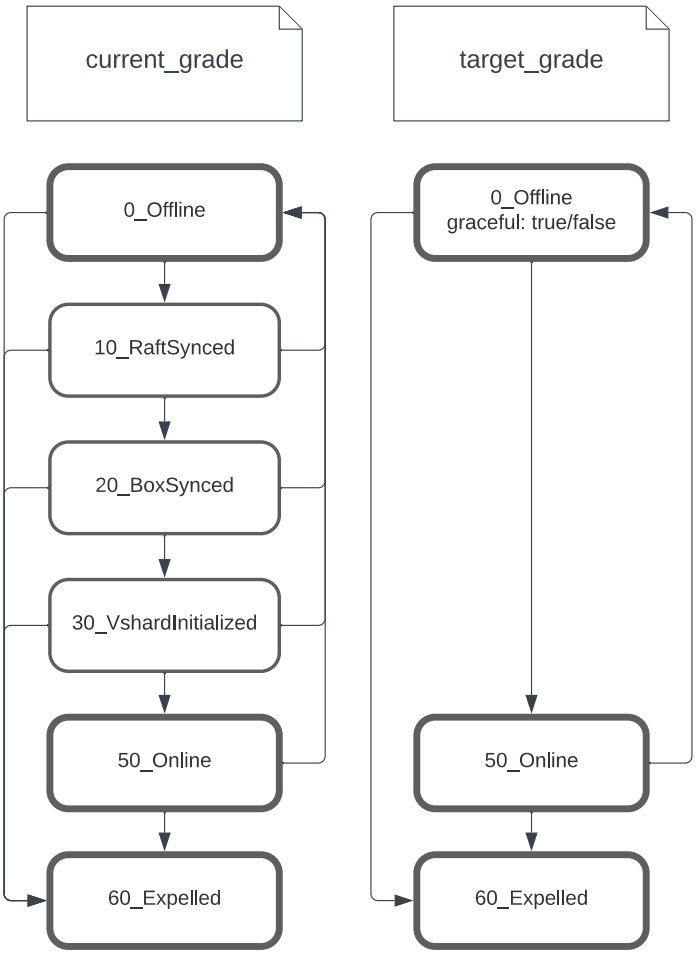
Приведением действительного к желаемому занимается специальный файбер на лидере — `governor_loop`. Он управляет всеми инстансами сразу.

С грейдом (как с текущим, так и с целевым) также всегда ассоциирована инкарнация (`incarnation`) — порядковое число, отражающее число попыток обработать данный инстанс со стороны файбера `governor_loop`. Это позволяет реагировать на ситуации, когда инстансы выходят из строя на какой-то период времени, после чего их необходимо снова привести в актуальное состояние.

На основе совокупности грейдов и их инкарнаций `governor_loop` на каждой итерации бесконечного цикла генерирует активности (activity) и пытается их организовать. Пока не организует, никаких других изменений в текущих грейдах не произойдет (но могут измениться целевые). Если активности завершатся ошибкой, то на следующей итерации они будут перевычислены с учетом новых целей.

Инкарнации грейдов вычисляются по следующему принципу. - Каждый раз когда `target_grade` инстанса получает значение `Online`, его инкарнация увеличивается на 1. - Все остальные изменения грейдов копируют инкарнацию с противоположного грейда, то есть при изменении `target_grade` инкарнация копируется с `current_grade`, при изменении `current_grade` — с `target_grade`.

Дальше перечислены активности, которыми занимается `governor_loop`, в том же порядке, в котором он к ним приступает.




*Instance states*

Ниже перечислены существующие варианты активностей, которые создает `topology_governor`.

### *1. Обновить состав голосующих / неголосующих инстансов*

Сначала нужно проверить необходимость менять конфигурацию Raft-группы, а именно — состав голосующих / неголосующих узлов (`voters` и `learners`).

Правила выбора новой конфигурации описаны в `picodata::governor::cc::raft_conf_change` и заключаются в следующем: - Любые инстансы, переходящие в грейд `Expelled`, удаляются из Raft-группы; - Голосущие инстансы, переходящие в грейд `Offline`, перестают быть голосующими (становятся `learners`) и для них находится замена; - Среди свежедобавленных инстансов с текущим грейдом `Online` подбирается необходимое количество голосующих инстансов (`voters`), остальные добавляются как `learners`;

По этим правилам создается `ConfChangeV2`, и, если он не пуст, отправляется в Raft. Далее нужно дождаться события `TopologyChanged`, которое будет послано в ответ на успешное применение новой конфигурации.

### *2. target_grade Offline / Expelled.*

Ниже рассмотрены два варианта вывода инстанса из строя: временный (`target_grade = Offline`) и постоянный (`target_grade = Expelled`). Перед тем как выключить инстанс, нужно убедиться, что кластер сможет продолжить функционировать без него.

Если уходит лидер Raft-группы, то есть инстанс, на котором в данный момент выполняется `governor_loop`, то он снимает с себя полномочия (делает `transfer_leadership`) и ждет смены Raft-статуса, дальше действовать будет кто-то другой.

Если уходит лидер своего репликасета, то происходят новые выборы такого лидера, после чего нужно дождаться соответствующей записи в спейс с репликасетами.

Далее следует обновить конфигурацию шардирования (`vshard`) на всех инстансах с ролями хранения данных (`storage`) и маршрутизации (`routers`), чтобы оповестить их об изменениях в топологии. Если это последний узел хранения в репликасете, ему будет выставлен вес 0.

Наконец, инстансу присваивается `current_grade`, соответствующей его целевому уровню.

### *3. target_grade: Online, current_grade: * -> RaftSynced*

Дальше начинается обработка инстансов, которых нужно привести в актуальное состояние. Это либо свежедобавленные инстансы, либо инстансы, которые были какое-то время неактивны.

Выбираем инстанс, либо имеющий `current_grade: Offline`, либо имеющий инкарнацию текущего грейда меньше, чем инкарнацию целевого.

На этом этапе мы синхронизируем Raft-журнал выбранных инстансов. Берем текущий `commit_index` лидера и дожидаемся, пока `commit_index` пира его не догонит. После этого присваиваем инстансу `current_grade = RaftSynced`.

### *4. target_grade: Online, current_grade: RaftSynced -> Replicated*

Этот этап отвечает за настройку репликации внутри одного репликасета, к которому относится выбранный инстанс.

Первым делом мы сообщаем всем инстансам репликасета, что необходимо применить новую конфигурацию репликации через `box.cfg { replication = ... }`. Однако, так как конфигурация кластера (в том числе и конфигурация репликасетов) распространяется между инстансами через Raft-журнал, необходимо убедиться что журнал у всех свежий. Для этого в запросе также передаем `commit_index`, которого пиры должны дождаться прежде чем выполнять сам запрос.

После этого инстансу, инициировавшему активность, присваивается `current_grade: Replicated`.

На этом же этапе добавляем запись в спейс с репликасетами, если ее там еще нет. При этом вес шардирования устанавливается в 0, если только это не первый репликасет в кластере.

Последнее что нужно сделать на этом этапе, это обновить значение `box.cfg { read_only }` в конфигурации лидера затронутого репликасета.

### *5. target_grade: Online, current_grade: Replicated -> ShardingInitialized*

На данном этапе настраивается шардирование всего кластера, поэтому запросы отправляются сразу всем инстансам.

Рассылаем всем запрос на обновление конфигурации шардирования (`vshard.router.cfg()` и `vshard.storage.cfg()`) опять вместе с `commit_index`, чтобы инстансы получили последние данные.

На этом этапе первый репликасет, наполненный до фактора репликации, запускает начальное распределение бакетов (`vshard.router.bootstrap`)

В конце этого этапа подсистема шардирования данных (`vshard`) на всех инстансах знает о топологии всего кластера, но на некоторых репликасетах вес все еще проставлен вес 0, поэтому данные на них ребалансироваться еще не будут.

### *6. target_grade: Online, current_grade: ShardingInitialized -> Online*

Этот этап нужен для того чтобы запустить ребалансировку данных на новые репликасеты. Для этого проверяем, есть ли у нас репликасеты с весом 0 и достигнутым фактором репликации. Если есть, то обновляем их вес и повторно обновляем конфигурацию шардирования на всем кластере, чтобы данные начали ребалансироваться.

## Минимальный вариант кластера

В данном разделе рассматриваются различные сценарии работы с кластером. Все они основаны на одном и том же принципе: запуске и объединении отдельных экземпляров Picodata в распределенный кластер. При этом сложность развертывания и поддержания работоспособности кластера зависит только от сложности его топологии.

Picodata может создать кластер, состоящий всего из одного экземпляра/инстанса. Обязательных параметров у него нет, что позволяет свести запуск к выполнению всего одной простой команды:

```
picodata run
```

Можно добавлять сколько угодно последующих инстансов — все они будут подключаться к этому кластеру. Каждому инстансу следует задать отдельную рабочую директорию (параметр `--data-dir`), а также указать адрес и порт для приема соединений (параметр `--listen`) в формате `<HOST>:<PORT>`. Фактор репликации по умолчанию равен 1 — каждый инстанс образует отдельный репликасет. Если для `--listen` указать только порт, то будет использован IP-адрес по умолчанию (127.0.0.1):

```
picodata run --data-dir i1 --listen :3301
picodata run --data-dir i2 --listen :3302
picodata run --data-dir i3 --listen :3303
```

Если не использовать параметр `cluster-id`, то по умолчанию кластер будет носить имя `demo`.

## Кластер на нескольких серверах

Выше был показан запуск Picodata на одном сервере, что удобно для тестирования и отладки, но не отражает сценариев полноценного использования кластера. Поэтому ниже будет показан запуск Picodata на нескольких серверах. Предположим, что их два: `192.168.0.1` и `192.168.0.2`. Порядок действий будет следующим:

На `192.168.0.1`:

```
picodata run --listen 192.168.0.1:3301
```

На `192.168.0.2`:

```
picodata run --listen 192.168.0.2:3301 --peer 192.168.0.1:3301
```

На что нужно обратить внимание:

Во-первых, для параметра `--listen` вместо стандартного значения `127.0.0.1` надо указать конкретный адрес. Формат адреса допускает упрощения — можно указать только хост `192.168.0.1` (порт по умолчанию `:3301`), или только порт, но для наглядности лучше использовать полный формат `<HOST>:<PORT>`.

Значение параметра `--listen` не хранится в кластерной конфигурации и может меняться при перезапуске инстанса.

Во-вторых, надо дать инстансам возможность обнаружить друг друга для того чтобы механизм discovery правильно собрал все найденные экземпляры Picodata в один кластер. Для этого в параметре `--peer` нужно указать адрес какого-либо соседнего инстанса. По умолчанию значение параметра `--peer` установлено в `127.0.0.1:3301`. Параметр `--peer` не влияет больше ни на что, кроме механизма обнаружения других инстансов.

Параметр `--advertise` используется для установки публичного IP-адреса и порта инстанса. Параметр сообщает, по какому адресу остальные инстансы должны обращаться к текущему. По умолчанию он равен `--listen`, поэтому в примере выше не упоминается. Но, например, в случае `--listen 0.0.0.0` его придется указать явно:

```
picodata run --listen 0.0.0.0:3301 --advertise 192.168.0.1:3301
```

Значение параметра `--advertise` анонсируется кластеру при запуске инстанса. Его можно поменять при перезапуске инстанса или в процессе его работы командой `picodata set-advertise`.

## Именование инстансов

Чтобы проще было отличать инстансы друг от друга, им можно давать имена:

```
picodata run --instance-id barsik
```

Если имя не дать, то оно будет сгенерировано автоматически в момент добавления в кластер. Имя инстанса задается один раз и не может быть изменено в дальнейшем (например, оно постоянно сохраняется в снапшотах инстанса). В кластере нельзя иметь два инстанса с одинаковым именем — пока инстанс живой, другой инстанс сразу после запуска получит ошибку при добавлении в кластер. Тем не менее, имя можно повторно использовать если предварительно исключить первый инстанс с таким именем из кластера.

## Проверка работы кластера

Каждый инстанс Picodata — это отдельный процесс в ОС. Для его диагностики удобно воспользоваться встроенной консолью, которая автоматически открывается после запуска инстанса (`picodata run ...`). Для диагностики всей Raft-группы (например, для оценки количества инстансов в кластере) выполните следующую команду:

```
box.space.raft_group:fselect()
```

Дополнительно, можно добиться ответа инстансов с помощью такой команды:

```
picolib.raft_propose_info("Hello, Picodata!")
```

В журнале каждого инстанса (по умолчанию выводится в `stderr`) появится фраза “Hello, Picodata!”

## Репликация и зоны доступности (failure domains)

Количество инстансов в репликасете определяется значением переменной `replication_factor`. Внутри кластера используется один и тот же `replication_factor`.

Управление количеством происходит через параметр `--init-replication-factor`, который используется только в момент запуска первого инстанса. При этом, значение из аргументов командной строки записывается в конфигурацию кластера. В дальнейшем значение параметра `--init-replication-factor` игнорируется.

По мере усложнения топологии возникает еще один вопрос — как не допустить объединения в репликасет инстансов из одного и того же датацентра. Для этого в Picodata имеется параметр `--failure-domain` — *зона доступности*, отражающая признак физического размещения сервера, на котором выполняется инстанс Picodata. Это может быть как датацентр, так и какое-либо другое обозначение расположения: регион (например, `eu-east`), стойка, сервер, или собственное обозначение (blue, green, yellow). Ниже показан пример запуска инстанса Picodata с указанием зоны доступности:

```
picodata run --init-replication-factor 2 --failure-domain region=us,zone=us-west-1
```

Добавление инстанса в репликасет происходит по следующим правилам:

- Если в каком-либо репликасете количество инстансов меньше необходимого фактора репликации, то новый инстанс добавляется в него при условии, что их параметры `--failure-domain` отличаются (регистр символов не учитывается).
- Если подходящих репликасетов нет, то Picodata создает новый репликасет.

Параметр `--failure-domain` играет роль только в момент добавления инстанса в кластер. Принадлежность инстанса репликасету впоследствии не меняется.

Как и параметр `--advertise`, значение параметра`--failure-domain` каждого инстанса можно редактировать, перезапустив инстанс с новыми параметрами.

Добавляемый инстанс должен обладать тем же набором параметров, которые уже есть в кластере. Например, инстанс `dc=msk` не сможет присоединиться к кластеру с `--failure-domain region=eu/us` и вернет ошибку.

Как было указано выше, сравнение зон доступности производится без учета регистра символов, поэтому, к примеру, два инстанса с аргументами `--failure-domain region=us` и `--failure-domain REGION=US` будут относиться к одному региону и, следовательно, не попадут в один репликасет.

## Динамическое переключение голосующих узлов в Raft (Raft voter failover)

Все узлы Raft в кластере делятся на два типа: голосующие (`voter`) и неголосующие (`learner`). За консистентность Raft-группы отвечают только узлы первого типа. Для коммита каждой транзакции требуется собрать кворум из `N/2 + 1` голосующих узлов. Неголосующие узлы в кворуме не участвуют.

Чтобы сохранить баланс между надежностью кластера и удобством его эксплуатации, в Picodata предусмотрена удобная функция — динамическое переключение типа узлов. Если один из голосующих узлов становится недоступным или прекращает работу (что может нарушить кворум в Raft), то тип `voter` автоматически присваивается одному из доступных неголосующих узлов. Переключение происходит незаметно для пользователя.

Количество голосующих узлов в кластере не настраивается и зависит только от общего количества инстансов. Если инстансов 1 или 2, то голосующий узел один. Если инстансов 3 или 4, то таких узлов три. Для кластеров с 5 или более инстансами — пять голосующих узлов.

## Удаление инстансов из кластера (expel)

Удаление — это принятие кластером решения, что некий инстанс больше не является участником кластера. После удаления кластер больше не будет ожидать присутствия инстанса в кворуме, а сам инстанс завершится. При удалении текущего лидера будет принудительно запущен выбор нового лидера.

### Удаление инстанса с помощью консольной команды

`picodata expel --instance-id <instance-id> [--cluster-id <cluster-id>] [--peer <peer>]`
где `cluster-id` и `instance-id` — данные об удаляемом инстансе, `peer` — любой инстанс кластера.

Пример:

`picodata expel --instance-id i3 --peer 192.168.100.123`
В этом случае на адрес `192.168.100.123:3301` будет отправлена команда `expel` с `instance-id = "i3"` и стандартным значением `cluster-id`. Инстанс на `192.168.100.123:3301` найдет лидера и отправит ему команду `expel`. Лидер отметит, что указанный инстанс удален; остальные инстансы получат эту информацию через Raft. Если удаляемый инстанс запущен, он завершится, если не запущен — примет информацию о своем удалении при запуске и затем завершится. При последующих запусках удаленный инстанс будет сразу завершаться.

### Удаление инстанса из консоли Picodata с помощью Lua API

В консоли запущенного инстанса введите следующее:

`picolib.expel(<instance-id>)`
например:

`picolib.expel("i3")`
Будет удален инстанс `i3`. Сам инстанс `i3` будет завершен. Если вы находитесь в консоли удаляемого инстанса — процесс завершится, консоль будет закрыта.

## Описание встроенных команд

Полный список аргументов доступен с помощью следующей команды:

```
picodata <command> [<params>]
```

### Описание команды `run`

Ниже приводится описание аргументов команды `run`.

`--advertise <[host][:port]>`{:name=‘advertise’} Адрес, по которому другие инстансы смогут подключиться к данному инстансу. По умолчанию используется значение из аргумента `--listen`. Аналогичная переменная окружения: `PICODATA_ADVERTISE`.

`--cluster-id <name>`{:name=‘cluster-id’} Имя кластера. Инстанс не сможет стать частью кластера, если у него указано другое имя. Аналогичная переменная окружения: `PICODATA_CLUSTER_ID`.

`--data-dir <path>`{:name=‘data-dir’} Директория, в которой инстанс будет сохранять свои данные для постоянного хранения. Аналогичная переменная окружения: `PICODATA_DATA_DIR`.

`-e, --tarantool-exec <expr>`{:name=‘tarantool-exec’} Данный аргумент позволяет выполнить Lua-скрипт на Tarantool

`--failure-domain <key=value>`{:name=‘failure-domain’} Список параметров географического расположения сервера (через запятую). Также этот аргумент называется *зоной доступности*. Каждый параметр должен быть в формате КЛЮЧ=ЗНАЧЕНИЕ. Также, следует помнить о том, что добавляемый инстанс должен обладать тем же набором доменов (т.е. ключей данного аргумента), которые уже есть в кластере. Picodata будет избегать помещения двух инстансов в один репликасет если хотя бы один параметр зоны доступности у них совпадает. Соответственно, инстансы будут формировать новые репликасеты. Аналогичная переменная окружения: `PICODATA_FAILURE_DOMAIN`.

`-h, --help`{:name=‘help’} Вывод справочной информации

`--init-replication-factor <INIT_REPLICATION_FACTOR>`{:name=‘init-replication-factor’} Число реплик (инстансов с одинаковым набором хранимых данных) для каждого репликасета. Аргумент используется только при начальном создании кластера и в дальнейшем игнорируется. Аналогичная переменная окружения: `PICODATA_INIT_REPLICATION_FACTOR`.

`--instance-id <name>`{:name=‘instance-id’} Название инстанса. Если этот аргумент не указать, то название будет сгенерировано автоматически. Данный аргумент удобно использовать для явного указания инстанса при его перезапуске (например, в случае его недоступности или при переносе в другую сеть). Аналогичная переменная окружения: `PICODATA_INSTANCE_ID`.

`-l, --listen <[host][:port]>`{:name=‘listen’} Адрес и порт привязки инстанса. По умолчанию используется localhost:3301 Аналогичная переменная окружения: `PICODATA_LISTEN`.

`--log-level <LOG_LEVEL>`{:name=‘log-level’} Уровень регистрации событий. Возможные значения: fatal, system, error, crit, warn, info, verbose, debug. По умолчанию используется уровень info. Аналогичная переменная окружения: `PICODATA_LOG_LEVEL`.

`--peer <[host][:port]>`{:name=‘peer’} Адрес другого инстанса. В данному аргументе можно передавать несколько значение через запятую. По умолчанию используется значение localhost:3301, т.е. без связывания с каким-либо другим инстансом. Указание порта опционально. Аналогичная переменная окружения: `PICODATA_PEER`.

`--replicaset-id <name>`{:name=‘replicaset-id’} Название целевого репликасета. Аналогичная переменная окружения: `PICODATA_REPLICASET_ID`

## Описание команды `tarantool`

Открывает консоль с Lua-интерпретатором, в котором можно взаимодействовать с СУБД аналогично тому как это происходит в обычной консоли Tarantool. Никакая логика Picodata поверх Tarantool не выполняется, соответственно, кластер не инициализируется и подключение к кластеру не производится. Запускается консоль Tarantool, встроенного в Picodata (но не установленного обычного Tarantool, если такой есть в системе).

## Описание команды `expel`

Исключает инстанс из кластера. Применяется чтобы указать кластеру, что инстанс больше не участвует в кворуме Raft.

Полный формат:

```
picodata expel --instance-id <instance-id> [--cluster-id <cluster-id>] [--peer <peer>]
```

Команда подключается к `peer` через протокол *netbox* и отдает ему внутреннюю команду на исключение `instance-id` из кластера. Команда отправляется в raft-лог, из которого затем будет применена к таблице инстансов и установит значение `target_grade=Expelled` для заданного инстанса. Затем через какое-то время *governor* возьмет в обработку этот `target_grade`, выполнит необходимые работы по отключению инстанса и установит ему значение `current_grade=Expelled`. Сам инстанс после этого остановится, его процесс завершится. В дальнейшем кластер не будет ожидать от этого инстанса участия в кворуме. Исключенный из кластера инстанс при попытке перезапуститься будет автоматически завершаться.

Параметр `cluster-id` проверяется перед добавлением команды в raft-лог.

Параметр `peer` — это адрес любого инстанса кластера. Формат: `[host]:port`. Может совпадать с адресом исключаемого инстанса.

Если исключаемый инстанс является текущим raft-лидером, то лидерство переходит другому инстансу.

Обратите внимание, что исключенный инстанс нужно снять из-под контроля супервизора.

Значение `instance-id` исключенного инстанса может быть использовано повторно. Для этого достаточно запустить новый инстанс с тем же `instance-id`.

## Примеры

Ниже приведены типовые ситуации и подходящие для этого команды.

1. На хосте с инстансом `i4` вышел из строя жесткий диск, данные инстанса утрачены, сам инстанс неработоспособен. Какой-то из оставшихся инстансов доступен по адресу `192.168.104.55:3301`.

   ```
   picodata expel --instance-id i4 --peer 192.168.104.9:3301
   ```
2. В кластере `mycluster` из 3-х инстансов, где каждый работает на своем физическом сервере, происходит замена одного сервера. Выключать инстанс нельзя, так как оставшиеся 2 узла кластера не смогут создать стабильный кворум. Поэтому сначала в сеть добавляется дополнительный сервер:

   ```
   picodata run --instance-id i4 --peer 192.168.104.1 --cluster-id mycluster
   ```
   Далее, если на сервере с инстансом i3 настроен автоматический перезапуск Picodata в Systemd или как-либо иначе, то его нужно предварительно отключить. После этого с любого из уже работающих серверов кластера исключается инстанс i3:

   ```
   picodata expel --instance-id i3 --cluster-id mycluster
   ```
   Указанная команда подключится к `127.0.0.1:3301`, который самостоятельно найдет лидера кластера и отправит ему команду на исключение инстанса `i3`. Когда процесс `picodata` на i3 завершится — сервер можно выключать.

# Пример работы с кластером Picodata

В данном разделе приведены практические примеры команд, которые помогут сделать первые шаги в управлении распределенным кластером Picodata. В частности, в данном разделе рассмотрены следующие вопросы:

- Запуск кластера
- Мониторинг состояния кластера
- Первые действия в только что созданном кластере
- Запись и чтение данных в кластере
- Балансировка данных в кластере

## Запуск кластера

Запуск кластера сводится к выполнению команды `picodata run` с нужным набором параметров для каждого инстанса (узла). Полный перечень возможных параметров запуска и их описание содержатся в подразделе Описание параметров запуска, а также в выводе команды `picodata run --help`. С точки зрения внутренней архитектуры, *кластер* корректно называть *Raft-группой* — в дальнейшем при мониторинге и управлении конфигурацией будет уместнее использовать именно этот термин. Для данного примера допустим, что в локальном кластере (127.0.0.1/localhost) будет 4 инстанса с фактором репликации равным 2, что означает наличие 2-х репликасетов. Запустим первый инстанс, указав необходимые параметры:

```
picodata run --init-replication-factor=2 --listen :3301 --data-dir=inst1
```

Следует обратить внимание на следующие моменты:

- Параметр `init-replication-factor` задается лишь один раз в момент создания кластера и дальше не требуется. Перезапускать данный инстанс в дальнейшем нужно без этого параметра.
- Параметр `listen` может содержать только номер порта (по умолчанию для первого инстанса используется 3301), что означает указание использовать *текущий* хост. В настоящем распределенном кластере указывать IP-адрес в данном параметре обязательно.
- Параметр `data-dir` указывает на директорию, в которой будут храниться персистентные данные инстансы (файлы `*.snap` и `*.xlog`). Если при первом запуске задать несуществующую директорию, то она будет автоматически создана.
- Будет создан кластер со стандартным названием `demo`, т.к. явно не указан параметр `cluster-id`.

Аналогично следует запустить остальные 3 инстанса, указав им отличные от 3301 разные порты и другие рабочие директории. В случае с кластером на удаленных узлах потребуется также указать данным инстансам параметр `peer`.

## Мониторинг состояния кластера

Для мониторинга состояния кластера удобно использовать команды, показывающие состояние Raft-группы, отдельных инстансов и собранных из них репликасетов. Для

использования указанных команд следует сначала подключиться к какому-либо инстансу с помощью команды `tarantoolctl connect`. Примеры команд и их выводов приведены ниже.

Узнать лидера Raft-группы, а также ID и статус текущего инстанса:

```
pico.raft_status()
```

Пример вывода:

```
---
- term: 2
  leader_id: 1
  raft_state: Leader
  id: 1
...
```

Просмотр состава Raft-группы и данных инстансов:

```
box.space.raft_group:fselect()
```

Пример вывода:

```
---
- - +----------+---------+-------+---------+---------+---------+---------+---------+---------+---------+
  - |instance_i|instance_|raft_id|peer_addr|replicase|replicase|commit_in|current_g|target_gr|failure_d|
  - +----------+---------+-------+---------+---------+---------+---------+---------+---------+---------+
  - |   "i1"   |"68d4a766|   1   |"localhos|  "r1"   |"e0df68c5|   12    |["Online"|["Online"|   {}    |
  - |   "i2"   |"24c4ac5f|   2   |"localhos|  "r1"   |"e0df68c5|   19    |["Online"|["Online"|   {}    |
  - |   "i3"   |"5d7a7353|   3   |"localhos|  "r2"   |"eff4449e|   28    |["Online"|["Online"|   {}    |
  - |   "i4"   |"826cbe5e|   4   |"localhos|  "r2"   |"eff4449e|   37    |["Online"|["Online"|   {}    |
  - +----------+---------+-------+---------+---------+---------+---------+---------+---------+---------+
...
```

Просмотр списка репликасетов, их веса и версии схемы данных:

```
box.space.replicasets:fselect()
```

Пример вывода:

```
---
- - +-------------+--------------------------------------+---------+------+----------------------+
  - |replicaset_id|           replicaset_uuid            |master_id|weight|current_schema_version|
  - +-------------+--------------------------------------+---------+------+----------------------+
  - |    "r1"     |"e0df68c5-e7f9-395f-86b3-30ad9e1b7b07"|  "i1"   |  1   |          0           |
  - |    "r2"     |"eff4449e-feb2-3d73-87bc-75807cb23191"|  "i3"   |  1   |          0           |
  - +-------------+--------------------------------------+---------+------+----------------------+
...
```

Эти и другие команды сведены в Bash-скрипт, который можно загрузить и выполнить для более удобного мониторинга кластера Picodata (например, командой `watch ./picodata-list.sh`).

Внешний вид выполняющегося скрипта показан ниже.

```
connected to localhost:3301
---
- instance_id: i1
- raft_state: Leader
- voters: [1, 2, 3]
- learners: [4]
- instances:
  - +----------+---------+-------+---------+---------+---------+---------+---------+---------+---------+
  - |instance_i|instance_|raft_id|peer_addr|replicase|replicase|commit_in|current_g|target_gr|failure_d|
  - +----------+---------+-------+---------+---------+---------+---------+---------+---------+---------+
  - |   "i1"   |"68d4a766|   1   |"localhos|  "r1"   |"e0df68c5|   12    |["Online"|["Online"|   {}    |
  - |   "i2"   |"24c4ac5f|   2   |"localhos|  "r1"   |"e0df68c5|   19    |["Online"|["Online"|   {}    |
  - |   "i3"   |"5d7a7353|   3   |"localhos|  "r2"   |"eff4449e|   28    |["Online"|["Online"|   {}    |
  - |   "i4"   |"826cbe5e|   4   |"localhos|  "r2"   |"eff4449e|   37    |["Online"|["Online"|   {}    |
  - +----------+---------+-------+---------+---------+---------+---------+---------+---------+---------+
- replicasets:
  - +-------------+--------------------------------------+---------+------+----------------------+
  - |replicaset_id|           replicaset_uuid            |master_id|weight|current_schema_version|
  - +-------------+--------------------------------------+---------+------+----------------------+
  - |    "r1"     |"e0df68c5-e7f9-395f-86b3-30ad9e1b7b07"|  "i1"   |  1   |          1           |
  - |    "r2"     |"eff4449e-feb2-3d73-87bc-75807cb23191"|  "i3"   |  1   |          1           |
  - +-------------+--------------------------------------+---------+------+----------------------+
...
```

Данный скрипт выполняет, в частности, следующие действия:

- При выполнении без аргументов подключается к первому локальному инстансу localhost:3301 с помощью консоли `tarantoolctl`. В качестве аргумента скрипту можно передать произвольное значение `<host:port>`.
- Выводит идентификатор (значение `instance_id`) текущего инстанса.
- Выводит статус текущего инстанса в Raft.
- Выводит количество голосующих/неголосующих узлов (`voters`/`learners`) в кластере.
- Выводит таблицы со списками инстансов и репликасетов.
- Позволяет узнать текущий и целевой уровень (`grade`) каждого инстанса, а также вес (`weight`) репликасета. Уровни отражают конфигурацию остальных инстансов относительно текущего, а вес репликасета — его наполненность репликами согласно фактору репликации.

## Создание схемы данных

Перед тем как начать пользоваться СУБД, необходимо создать таблицу, которая в терминологии Tarantool называется `space`. Таблица является необходимым элементом схемы данных, распространяемой на все узлы кластера. Каждое действие по изменению схемы данных в Picodata называется *миграцией*. Иными словами, миграция — это переход кластера на использование более новой схемы данных. Любое действие по созданию/изменению/удалению таблиц, работы с индексами хранения и т.д. является изменением схемы данных. Каждое изменение инкрементирует версию схемы данных в кластере.

После подключения к инстансу кластера посредством утилиты `tarantoolctl`, начальным действием в пустом кластере будет добавление первого `space`. Пусть в нем будет два поля: идентификатор записи и идентификатор бакета, в которой эта запись хранится:

```
pico.add_migration(1, [[
CREATE TABLE "test" (
"id" int,
"bucket_id" unsigned,
PRIMARY KEY ("id")
);
]])
```

На данном этапе схема данных существует лишь локально, в коллекции текущего инстанса. Посмотреть доступные инстансу схемы данных можно командой `box.space.migrations:fselect()`. Результат будет выглядеть следующим образом:

```
localhost:3301> box.space.migrations:fselect()
---
- - +-----+-----------------------------------------------------------------------------------------+
- | id  |                                          body                                           |
- +-----+-----------------------------------------------------------------------------------------+
- |  1  |"CREATE TABLE \"test\" (\n\"id\" int,\n\"bucket_id\" unsigned,\nPRIMARY KEY (\"id\")\n);\n"|
- +-----+-----------------------------------------------------------------------------------------+
...
```

Теперь можно применить схему в рамках кластера, введя команду `pico.push_schema_version(1)`. Для того чтобы посмотреть параметры созданного `space` следует использовать команду `box.space.test`, где *test* — название таблицы. Также следует обратить внимание, что в выводе команды `box.space.replicasets:fselect()` обновится номер текущей схемы в кластере:

```
---
- - +-------------+--------------------------------------+---------+------+---------------------+
  - |replicaset_id|           replicaset_uuid            |master_id|weight|current_schema_version|
  - +-------------+--------------------------------------+---------+------+---------------------+
  - |    "r1"     |"e0df68c5-e7f9-395f-86b3-30ad9e1b7b07"|  "i1"   |  1   |          1          |
  - |    "r2"     |"eff4449e-feb2-3d73-87bc-75807cb23191"|  "i3"   |  1   |          1          |
  - +-------------+--------------------------------------+---------+------+---------------------+
...
```

В дальнейшем каждое изменение схемы данных в кластере будет приводить к увеличению этого номера.

## Вызов функций записи и чтения из БД

Для того, чтобы в таблицу/space можно было записывать данные, требуется сначала создать индекс БД. Для этого создадим еще одну миграцию схемы данных:

```
pico.add_migration(2, [[CREATE INDEX "bucket_id" on "test" ("bucket_id");]])
pico.push_schema_version(2)
```

После этого в таблицу можно вставлять строки, используя функцию записи из состава библиотеки `vshard`:

```
vshard.router.callrw (1, "box.space.test:insert", {{1, 1}})
```

Здесь первая и третья 1 — номер бакета, вторая — номер записи. Можно делать множество записей с разными номерами в один и тот же бакет. Пример для 4-й записи в 2000-м бакете:

```
vshard.router.callrw (2000, "box.space.test:insert", {{4, 2000}})
```

Просмотр сделанной записи:

```
vshard.router.callro (2000, "box.space.test:select")
```

## Запись и чтение данных

Для того чтобы записать в БД какие-либо настоящие данные (например, текстовую строку), нам потребуется создать новый `space` с еще одним полем для хранения такого текста, а также новым индексом. Это означает проведение еще двух миграций схемы данных.

Добавим `space` с названием `test1`:

```
pico.add_migration(3, [[
CREATE TABLE "test1" (
"id" int,
"bucket_id" unsigned,
"text" string,
PRIMARY KEY ("id")
);
]])
pico.push_schema_version(3)
```

Создадим на нем индекс:

```
pico.add_migration(4, [[CREATE INDEX "bucket_id" on "test1" ("bucket_id");]])
pico.push_schema_version(4)
```

Для просмотра всех полей таблицы, включая текстовые, подойдет следующая команда:

```
box.space.test1:format()
```

Пример записи текстовой строки:

```
vshard.router.callrw (1, "box.space.test1:insert", {{1, 1, "Sample text"}})
```

Проверка:

```
vshard.router.callro (1, "box.space.test1:select")
```

## Балансировка данных

Относительно бакетов в Picodata используются умолчания, принятые в СУБД Tarantool, согласно которым в кластере всегда доступны 3000 бакетов. Размер бакета динамичен: он определяется размером хранимых в нем данных. Бакеты равномерно распределяются между репликасетами. В приведенном здесь примере кластера из двух репликасетов, один из них хранит диапазон бакетов от 0 до 1500, а второй — от 1501 до 3000.

Для того чтобы просмотреть хранящиеся в текущем репликасете бакеты, используйте следующую команду:

```
box.space.test1:fselect()
```

Соответственно, если нужного бакета в списке нет, то он хранится в другом репликасете, и данную команду нужно выполнять на нем. Просмотреть список бакетов на текущем инстансе можно так:

```
box.space._bucket:fselect()
```

Балансировка данных в Picodata происходит автоматически при изменении конфигурации кластера, например при добавлении новых инстансов. Во время балансировки изменяется распределение бакетов между репликасетами. К примеру, если в кластере добавится новый полный (с весом 1) репликасет, то часть бакетов автоматически переедет на него. Это можно будет заметить при выполнении команды `box.space._bucket:fselect()`.