Научная статья

УДК 004.658.6
DOI 10.25205/1818-7900-2025-23-2-29-42

Алгоритм проверки предусловий корректности запуска репликации в отказоустойчивом кластере PostgreSQL

Андрей Сергеевич Рудометов
Михаил Валерьевич Рутман

Новосибирский государственный университет
Новосибирск, Россия

a.rudometov@g.nsu.ru
m.rutman@g.nsu.ru

Аннотация

Традиционно отказоустойчивые кластеры СУБД на базе PostgreSQL строятся с использованием механизма потоковой репликации, передающего файлы журнала предзаписи между узлами. При запуске репликации выполняются проверки лишь на целостность файлов журнала. Поэтому при вводе в кластер узлов после отработки отказа и настройки репликации можно получить резервный узел с данными, отличными от основного, либо выходящий из строя при попытке запуска или перезапуска. Существующие системы высокой доступности вынуждены требовать длительного процесса пересоздания узла в данных сценариях.

В работе предложен алгоритм, блокирующий запуск репликации в сценариях, когда это заведомо приведет к потере идентичности данных или сбоям при восстановлении узла. Для этого перед запуском выполняется проверка предусловия корректности, основанных на сборе и сравнении информации о состоянии журналов предзаписи узлов кластера. После блокировки возможна автоматическая синхронизация узлов для последующего корректного запуска репликации.

Предложен способ встраивания алгоритма в СУБД, проведено тестирование на различных конфигурациях с эмуляцией сбоев. При минимальных накладных расходах алгоритм предотвращает последствия некорректного запуска репликации в большинстве случаев.

Ключевые слова

репликация, отказоустойчивый кластер СУБД, высокая доступность, предсказание сбоев, PostgreSQL

Для цитирования

Рудометов А. С., Рутман М. В. Алгоритм проверки предусловий корректности запуска репликации в отказоустойчивом кластере PostgreSQL // Вестник НГУ. Серия: Информационные технологии. 2025. Т. 23, № 2. С. 29–42. DOI 10.25205/1818-7900-2025-23-2-29-42

Preconditions-Based Algorithm for Safe Start of Replication in Fault-Tolerant PostgreSQL Cluster

Andrey S. Rudometov, Mikhail V. Rutman

Novosibirsk State University,
Novosibirsk, Russian Federation

a.rudometov@g.nsu.ru
m.rutman@g.nsu.ru

Abstract

Traditionally, fault-tolerant DBMS clusters using PostgreSQL or derivatives are built on replication machinery, operated via write-ahead log shipping. Default checks are aimed only at preserving the integrity of received records. In certain conditions replication start can lead to standby cluster node having data different from other nodes, or being unable to finish startup procedures. Existing high availability systems are forced to cope with the problem through recreating such nodes from backups, which is usually costly in terms of recovery time.

To address this issue, we propose an algorithm to prevent replication start when it is guaranteed to lead to data differences or node startup failure. For detection of such cases node collects information about write-ahead logs in the cluster and performs additional checks. If replication was blocked, automatic node synchronization for consequent replication start is available.

We have tested the algorithm on various real-world cluster configurations with simulated failures, and the experimental results indicate that algorithm substantially reduces the chance of nodes being non-eligible to restart.

Keywords

replication, fault-tolerant DBMS cluster, high availability, failure prediction, PostgreSQL

For citation

Aleeva V. N., Kuznetsov E. K. Effective implementation of sorting algorithms using the concept of Q-determinant. *Vestnik NSU. Series: Information Technologies*, 2025, vol. 23, no. 2, pp. 29–42 (in Russ.) DOI 10.25205/1818-7900-2025-23-2-29-42

Введение

Репликация широко используется в распределенных системах как простой способ обеспечения избыточности с минимальной задержкой. Распределенные СУБД, в частности, основанные на PostgreSQL решения, не являются исключением; большинство программного обеспечения для управления отказоустойчивыми кластерами СУБД на базе PostgreSQL, например: patroni, pg_auto_failover, stolon, PostgresPro BiHA, pgEdge, полагается на встроенный в PostgreSQL механизм репликации для поддержания актуальной копии данных на резервных узлах кластера [1].

Потоковая репликация в PostgreSQL передает между узлами кластера файлы журнала предзаписи, работая при этом параллельно процессу, который применяет их содержимое в процессе запуска узла или его функционирования как резервного. Поскольку ожидается, что узлы имеют полностью идентичные журналы и данные, никаких дополнительных проверок при копировании и возможной перезаписи файлов журнала не выполняется.

С другой стороны, при функционировании в составе отказоустойчивого кластера и после реакции на отказы или изменения топологии кластера условие идентичности журналов узлов может нарушаться. Запуск репликации между узлами с разными журналами и/или разными текущими позициями применения записей из них может привести к различным сбоям узла. Сбои могут быть как легко наблюдаемые, например, резервный узел, на котором не применяются реплицированные данные, или узел, не обрабатывающий любые запросы к нему из-за невозможности окончить процедуру запуска, так и практически не диагностируемые. В качестве примера последнего можно привести возможность получить резервный узел с отличными от остальных данными, но нормально применяющий реплицированный журнал – подобная

ситуация будет обнаружена только при нарушении какой-либо проверки целостности при обращении к некорректным данным, если таковая проверка вообще существует.

Аппаратные и программные сбои могут приводить к выходу узлов из отказоустойчивого кластера. После нейтрализации некоторых сбоев, например, потери сетевой связности между узлами или отключения питания, узел может немедленно либо после минимального вмешательства войти в состав кластера и участвовать в резервировании данных. Для других видов сбоев, например, безвозвратной потери хранимых данных или полного выхода из строя всего центра обработки данных, восстановление и возврат узла в кластер будут невозможны.

Сбои, вызванные запуском репликации между узлами с разными журналами, сейчас тоже требуют пересоздания узла, поскольку не существует универсального механизма исправления последствий таких сбоев – пример с отличными данными это хорошо показывает. Поскольку полное пересоздание подразумевает восстановление узла из резервной копии, т. е. как минимум копирование всей базы данных и ожидание завершения репликации разницы с актуальными данными, то очевидно, что данный способ ввода узла в кластер много дольше обычного запуска узла. Чем дольше восстановление после отказа, тем выше риск потери данных из-за повторных отказов [2].

С другой стороны, на узлах кластера содержится вся необходимая информация для предсказания и недопущения сбоев после запуска репликации. Эта информация не используется ни встроенным механизмом репликации, из-за предположения об идентичности журналов, ни внешними решениями для построения кластеров, из-за трудности доступа к внутренней информации СУБД.

В работе предложен алгоритм принятия решения о безопасности запуска репликации. Алгоритм блокирует автоматический запуск репликации; после получения информации о состояниях журнала других узлов кластера и проверки набора предусловий либо разрешает репликацию, либо оставляет корректно работающий, но не подключенный к остальному кластеру узел. Во втором случае предлагается либо вмешательство системного администратора, либо автоматическая синхронизация узла с одним из узлов кластера посредством утилиты pg_rewind. Такая синхронизация занимает больше времени, чем репликация, но много меньше, чем пересоздание узла.

Предложенный алгоритм не зависит от конкретной архитектуры какого-либо решения для построения отказоустойчивого кластера и может использоваться в уже существующих системах после внесения правок в ядро СУБД и обновления узлов.

Сильное зацепление репликации и запуска узла

Надежное функционирование СУБД PostgreSQL обеспечивается использованием журнала предзаписи. Для описания предлагаемого алгоритма и решаемых им проблем достаточной абстракцией будет пронумерованная последовательность записей, хранимая в последовательности сегментных файлов. Номер следующей записи на единицу больше номера предыдущей. Новые записи добавляются в последний сегментный файл последовательности. При заполнении сегментного файла создается новый, последовательность записей продолжается в нем (рис. 1).

Исторически репликация не проектировалась частью сервера баз данных (или узла, в терминологии данной работы) PostgreSQL; вместо этого архитектура была сфокусирована на надежном механизме восстановления после сбоев, позволяя продолжать обслуживание клиентов после неожиданных остановов или перезапусков СУБД [3–5]. Когда была признана необходимость данного механизма и начали появляться сторонние расширения, реализующие логику передачи журнала предзаписи, например Slony-I [6], реализация встроенного механизма ре-

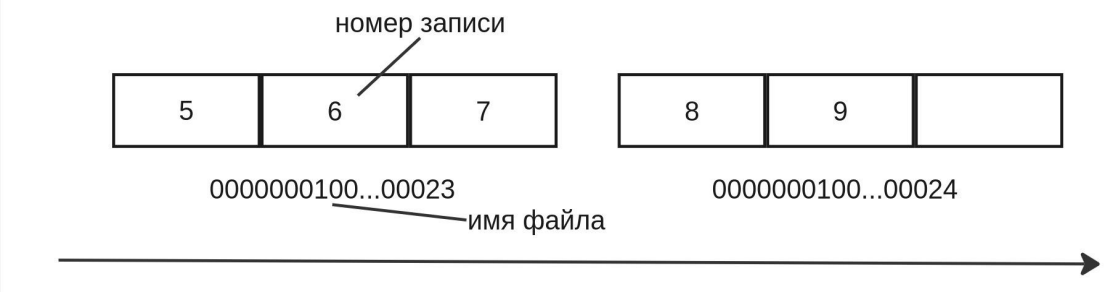

Рис. 1. Записи журнала предзаписи в сегментных файлах
Fig. 1. Write-ahead log records in segment files

пликации была добавлена в PostgreSQL 9.0 как расширение к существующей логике восстановления после сбоя [7].

Сервер PostgreSQL работает в одном из двух режимов: восстановления (режиме резерва) либо обычном. В обычном режиме обрабатываются все запросы клиентов, в режиме восстановления – только запросы на чтение и только после достижения СУБД целостного состояния.

После каждого запуска узел PostgreSQL переходит в режим восстановления и последовательно читает записи журнала предзаписи и применяет их согласно протоколу REDO. Чтение останавливается при достижении записи с целевым номером, после этого узел может перейти из режима восстановления в обычный режим и начать обрабатывать клиентские запросы.

Если узел не был остановлен с помощью команды администратора или используемых механизмов автоматизации, т. е. останов был неожиданным и не оставил записи в управляющих файлах СУБД, целевым номером записи считается номер последней из всех присутствующих в журнале узла на момент запуска.

Механизм репликации расширяет данную логику, добавляя процесс, способный получать записи журнала с других узлов и сохранять как часть журнала узла, в то время как узел продолжает оставаться в режиме восстановления и применять новые записи. В случае достижения конца журнала и отсутствия новых записей узел ожидает их появления.

Данный процесс, *walreceiver*, является частью довольно простого механизма – соответствующий ему процесс *walsender* на узле-источнике репликации в цикле проверяет разницу между номером последней отправленной записи и номером последней созданной и при необходимости пересылает составляющие разницу записи на узел-получатель, где они записываются в соответствующий сегментный файл журнала узла. Если в журнале узла-получателя уже были файлы или записи с такими же номерами, они будут перезаписаны, поскольку при нормальном функционировании репликации ожидается, что узлы имеют одинаковые журналы с единственной разницей в отставании содержимого одного от другого.

Если при попытке применения перезаписанной записи возникает ошибка, то такая ошибка заблокирует применение последующих записей, а при перезапуске узла не даст ему окончить восстановление – оно остановится на той же самой ошибке, оставляя узел недоступным для управления посредством SQL-интерфейса.

Хотя теоретически возможно вручную исправить содержимое записей журнала, отсутствие готовых инструментов для этого потребует сложного анализа бинарных файлов с высокой вероятностью ошибки, что делает метод неприменимым на практике.

Линии времени

В качестве части механизма встроенной репликации PostgreSQL предлагает возможность восстановления на момент времени (PITR). Если необходимые записи журнала еще не были

удалены или были восстановлены из архива, процесс восстановления может закончиться на заданной администратором целевой записи, соответствующей заданному времени, идентификатору транзакции и т. д. Поскольку записи, добавленные после выбранной целевой, все еще остаются в каталоге с файлами журнала, а после восстановления до заданной цели узел может перейти в обычный режим и начать добавлять с этого момента новые записи, то для дифференциации «старых» и «новых» записей была введена концепция линий времени (*timeline*) [8]. Линия времени с определенным номером объединяет записи, сделанные после последнего перехода узла в обычный режим.

Номер линии времени используется при именовании сегментных файлов журнала, что позволяет легко отличать записи, сделанные в разные линии времени. После каждого окончания восстановления номер линии времени инкрементируется, генерируя копию последнего сегментного файла журнала, но с увеличенным номером линии времени в имени. Информация о новой линии времени сохраняется в записи об окончании режима восстановления (*END_OF_RECOVERY, EOR* на схеме). Все новые записи будут записаны в этот и последующие файлы (рис. 2).

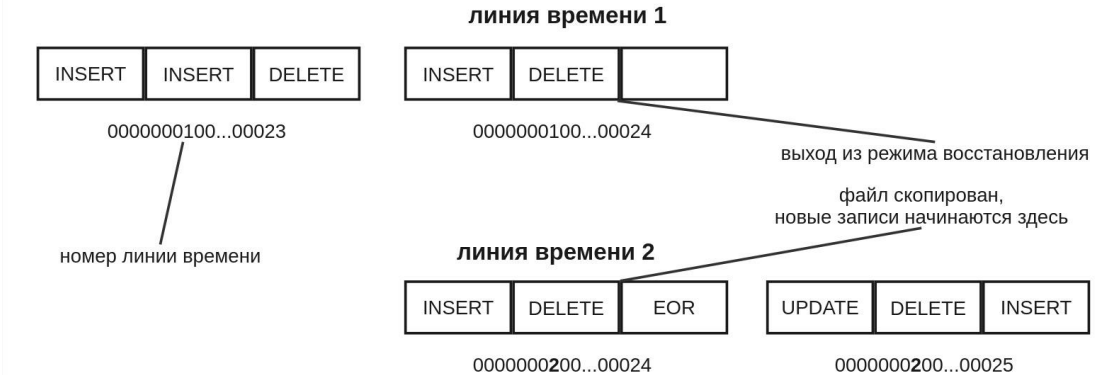

Рис. 2. Сегментные файлы после увеличения номера линии времени
Fig. 2. Segment files after timeline increment

Кроме того, механизм репликации поддерживает *файлы истории* – по одному для каждой линии времени вплоть до текущей. Файл содержит информацию о номерах записей журнала, на которых происходили все переходы на новую линию времени от первой до соответствующей файлу.

Анализ этих файлов для последней линии времени узла позволяет проводить сравнение журналов узлов без их чтения.

Синхронизация узлов: pg_rewind

Пусть синхронизация данных на узлах PostgreSQL при помощи репликации недоступна: из-за сбоев на уровне кластера и возможных split-brain [9] ситуаций история линий времени узлов различается, а значит, различается и содержимое журналов. В таком случае все еще остается способ синхронизации узлов более быстрой, чем инициализация нового из резервной копии [10].

Узел может быть синхронизирован с другим при помощи стандартной утилиты pg_rewind. Утилита ищет последнюю одинаковую запись в двух журналах, затем составляет список измененных последующими записями журнала файлов на узле-получателе и копирует соответствующие файлы с узла-источника [11]. После копирования для восстановления целостности данных необходимо применить все записи журнала, начиная с последней одинаковой.

Кроме того, при запуске узла-получателя после синхронизации необходимо реплицировать и применить все записи журнала, созданные на узле-источнике между моментами начала и окончания процесса копирования. Очевидно, что в противном случае целостность данных может быть нарушена: пусть изменение затронуло уже скопированный файл A и еще не скопированный файл B, тогда после синхронизации узел-получатель будет содержать только часть изменения в файле B.

Пример некорректного запуска репликации

Приведем пример сценария, который приводит к расхождению данных на узлах. Пусть узел A обрабатывает запросы клиентов (является ведущим узлом), узел B – его резервный, получает реплицированные записи и применяет их, находясь в режиме восстановления (рис. 3). Указатели replay_lsn, flush_lsn соответствуют номерам последней примененной и последней реплицированной с A записи. Пусть теперь связь между узлами теряется, причем на A остаются еще не реплицированные записи (на схеме выделены темно-серым фоном).

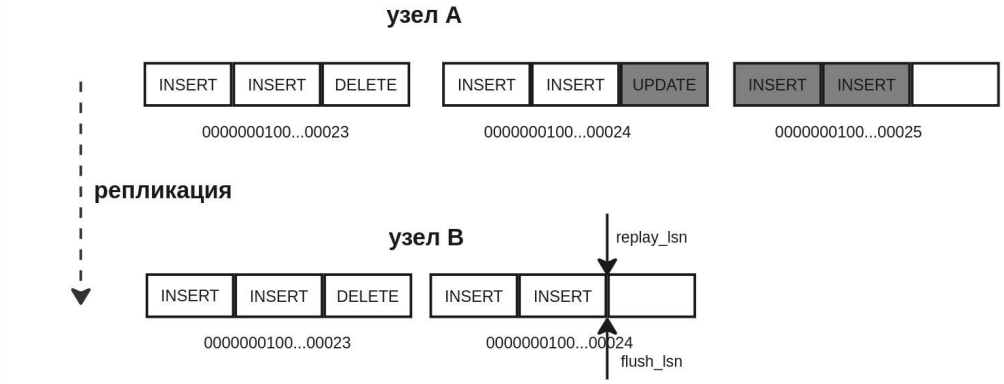

Рис. 3. Состояние журналов узлов до некорректного запуска репликации
Fig. 3. Node's logs before incorrect replication start

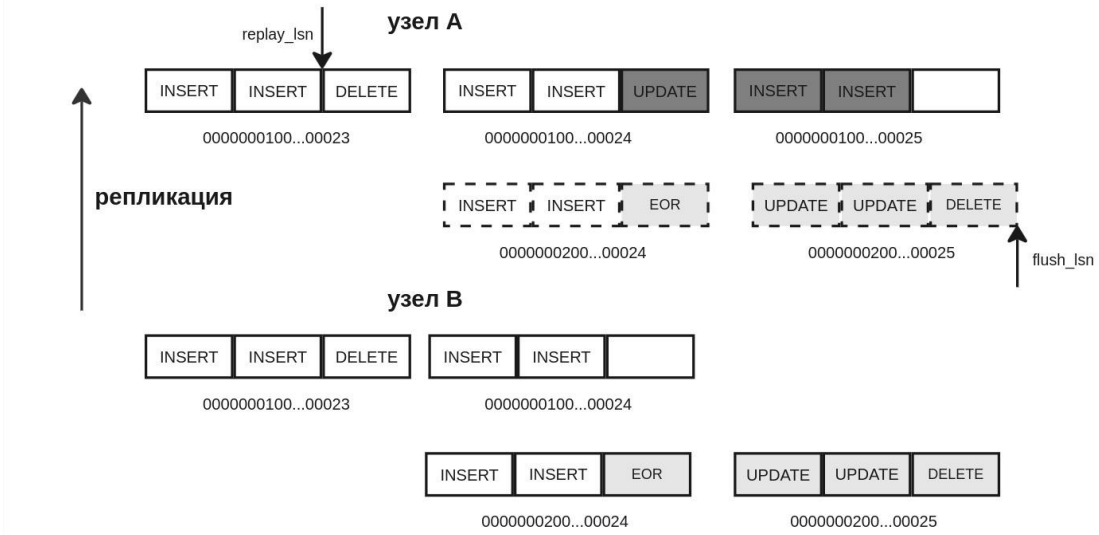

Рис. 4. Состояние журналов узлов после некорректного запуска репликации
Fig. 4. Node's logs after incorrect replication start

Если узлы A и B были частью отказоустойчивого кластера, то вместо A ведущим будет назначен один из резервных узлов. Пусть, не ограничивая общности, новым ведущим оказался B. Теперь, если мы хотим вернуть A в кластер как резервный узел, необходимо перезапустить его в режиме восстановления и настроить репликацию с B на A (рис. 4). На схеме светло-серым фоном выделены записи, созданные на B во время его работы ведущим. При запуске репликации с B на A будут переданы и записаны в журнал записи из файлов 24 и 25 второй линии времени – результат записи выделен пунктиром.

Тогда набор данных узла A будет зависеть от взаимного положения указателей применения и репликации записей. В данном примере предполагается, что перезапись происходит до того, как было начато применение перезаписываемых записей (replay_lsn < flush_lsn). Нетрудно заметить, что на узле A будет содержаться результат применения записей с темно-серым фоном, в частности файла 25 первой линии времени, а на всех остальных узлах кластера его не будет.

Обзор литературы

СУБД PostgreSQL не проектировалась с учетом использования в отказоустойчивых кластерах: наглядным примером является использование примитивного последовательного REDO при запуске. Решения для построения отказоустойчивых кластеров работают с предоставляемыми СУБД интерфейсами и не вмешиваются в логику работы с данными и журналом предзаписи, поэтому решения данной проблемы со стороны СУБД ранее не предлагалось. Однако область ускорения восстановления данных уже исследовалась ранее, и некоторые идеи могут быть применены в данной работе.

В работе [12] предлагается новый алгоритм мгновенного восстановления, основная идея которого заключается в поиске изменений, REDO которых может быть выполнено «ленивым» образом. Таким образом, эти изменения могут быть применены по необходимости (при доступе к соответствующим им данным) после приведения узла в состояние доступности. В то время как данная идея не может быть легко интегрирована в существующую логику восстановления СУБД, отложить применение реплицированных с других узлов изменений относительно применения уже имеющихся полезно для достижения поставленной цели.

В исследованиях онлайн-восстановления (online recovery [13]), «способности восстановления узла после сбоя, пока другие узлы исполняют запросы клиентов», авторы в основном фокусируются на использовании существующих внешних интерфейсов PostgreSQL, таких как восстановление их архива или восстановление на момент времени (PITR) [13; 14]. В то время как данные методы могут помочь избежать части сценариев, приводящих к невозможности запуска после сбоя репликации, при правильной конфигурации – они не универсальны, и такие побочные эффекты не являлись целью исследований.

Пока инкрементальное копирование, добавленное в PostgreSQL 17, было в стадии разработки, было предложено и реализовано несколько внешних решений, обычно полностью основанных на управлении WAL для снижения накладных расходов на чтение и запись – пример такого подхода демонстрируется в работе [15]. Хотя общая идея исследования подходит как инструмент решения обозначенной проблемы, внешние относительно СУБД решения не имеют доступа к полной информации о ее состоянии и способны использовать меньше методов управления этим состоянием.

Предлагаемый алгоритм

Основная идея алгоритма заключается в запрете запуска репликации через блокировку функций, приводящих к запуску процесса *walreceiver*. Запрет поддерживается, как минимум,

пока СУБД не будет уверена, что репликация с узла-источника согласно ее конфигурации не приведет к сбою.

Обычно *walreceiver* запускается без каких-либо проверок того, что именно будет записано в каталог, содержащий файлы журнала.

После установки блокировки алгоритм ожидает получения информации о состоянии журнала узла-источника, затем выполняет проверку предусловий запуска репликации и соответствующие действия согласно матрице решений.

Необходимая для проверки предусловий информация о состоянии журналов узла-источника и узла-получателя:

1) файл истории текущей линии времени узла-источника;
2) файл истории текущей линии времени узла-получателя;
3) номера последней примененной и последней записанной на диск записи журнала узла-источника;
4) номера последней примененной и последней записанной на диск записи журнала узла-получателя;
5) точка расхождения историй линий времени узла-источника и узла-получателя.

В решениях для построения отказоустойчивых кластеров для передачи информации о состоянии часто используется механизм heartbeat-сообщений, отправляемых периодически с заданным интервалом и содержащих краткий статус узла для принятия решений об изменении конфигурации кластера какими-либо внешними модулями управления.

История и номера транзакций узла-источника (пп. 1, 3 списка необходимой информации) могут быть встроены в такие сообщения, и тогда узлу-получателю нужно только дождаться установки соединения с узлом-источником и получения первого такого сообщения. Для эталонной реализации алгоритма реализовано расширение PostgreSQL, обеспечивающее соединение узлов кластера «каждый-с-каждым» и аналогичную рассылку heartbeat-сообщений с необходимыми параметрами и настраиваемой периодичностью. Расширение запрашивает у СУБД номер последней записи перед каждой отправкой, а содержимое файла истории текущей линии времени кэшируется и обновляется только при смене линии времени.

Для узла-получателя известен номер текущей линии времени, поэтому файл истории (п. 2) достаточно прочитать из каталога файлов журнала.

Получение информации для п. 4 сложно, поскольку во время принятия решения узел обязан находиться в состоянии восстановления, а в таком состоянии СУБД не имеет штатных средств для определения номера последней записи в журнале, т. е. последней записанной на диск. Состояние восстановления подразумевает, что выполнение записей последовательно и бесконечно, и в любой момент они могут быть добавлены в конец журнала – процессами репликации, при восстановлении из архива журнала или вручную администратором. Поэтому узлом в состоянии восстановления отслеживается только номер последней примененной записи.

В то время как возможна разработка метода надежного хранения необходимого номера, в предлагаемом алгоритме выбран более простой подход – узел должен завершить восстановление после сбоя до принятия решения согласно алгоритму, т. е применить все записи, содержащиеся в каталоге журнала, и перейти к ожиданию появления новых. Таким образом, номера последней примененной и последней записанной на диск записи совпадут.

Пусть в обеих историях линий времени двух узлов присутствует запись об окончании времени t, причем номера последних записей в них различаются на двух узлах. Тогда назовем точкой расхождения (п. 3) пару (t, $elsn$) такую, где t – наименьшая из удовлетворяющих условию, а $elsn$ – наименьший из двух различных номеров последней записи, соответствующих t. Из существования точки расхождения двух историй следует неодинаковость записей после точки расхождения, что приведет к сбоям после запуска репликации между узлами с такими историями.

Точка расхождения может быть найдена путем сопоставления обеих линий времени. Достаточно начать с линии времени 1 и сравнивать номера последних записей в линиях

времени с одинаковым номером, пока не будет достигнут конец одного или обоих файлов истории.

Матрица решений

Обозначим узел-получатель, стремящийся закончить процесс восстановления и войти в состав кластера СУБД, как узел A, а узел-источник, с которого ему необходимо реплицировать недостающие данные, как узел B.

Соответствующие номера текущей линии времени обозначим *a_tli* и *b_tli; m_tli = min(a_tli, b_tli*). Определим *elsn(tli)* как номер последней записи в линии времени.

Значение *elsn* для предыдущих линий времени может быть получено посредством чтения файлов истории. Для текущей линии времени в ее файле истории нет номера последней записи, только номер записи, на котором закончилась предыдущая линия времени. Поэтому доопределим *elsn* для текущей линии времени как номер последней примененной или записанной на диск записи, в зависимости от режима, в котором находится узел B.

Отметим также, что такое доопределение не делает корректным понятие точки расхождения для текущей линии времени (если *a_tli = b_tli*), поскольку она еще не завершена посредством перехода на следующую. В таком случае есть опасность посчитать точкой расхождения согласно ее определению сценарий, когда один из журналов имеет больше записей, но общие записи журналов совпадают, т. е. после запуска репликации журналы станут одинаковыми.

Зная обе текущие линии времени и их *elsn*, можно выделить ситуации, в которых запускать репликацию точно нельзя:

1. Если номер линии времени узла A больше, то очевидно, что на нем есть данные, которых нет на узле B. Если мы разрешим репликацию, то какие-то записи журнала будут перезаписаны, что с высокой вероятностью приведет к сбою.
2. Аналогично, если линии времени узлов совпадают, но *elsn(a_tli) > elsn(b_tli)*, существует хотя бы одна запись, встречающаяся только в журнале узла A.
3. Если номер линии времени A меньше такового у B, а *elsn(a_tli)*, наоборот, больше *elsn(b_tli)*, значит, *(a_tli, elsn(b_tli))* – точка расхождения.
4. Для линий времени, предшествующих *m_tli*, существует точка расхождения.

В данных случаях запуск репликации возможен, только если синхронизировать данные и журнал узла-получателя с узлом-источником, в предлагаемом алгоритме используется синхронизация при помощи pg_rewind.

В остальных случаях матрица принятия решений допускает запуск репликации, как показано на рис. 5.

	a_tli < b_tli	**a_tli = b_tli**	**a_tli > b_tli**
elsn(a_tli) < elsn(m_tli)	можно	можно	нельзя/ синхронизация
elsn(a_tli) < elsn(m_tli)	можно	можно	нельзя/ синхронизация
elsn(a_tli) < elsn(m_tli)	нельзя/ синхронизация	нельзя/ синхронизация	нельзя/ синхронизация

Рис. 5. Матрица принятия решений
Fig. 5. Decision matrix

В случае совпадения номеров линий времени нельзя говорить о полной уверенности отсутствия расхождения журналов узлов.

Поскольку алгоритм сравнивает только номера позиций, но не содержимое записей журнала, возможны ситуации, когда на узле A меньше или столько же записей, но их содержимое отлично от такового в части записей узла B.

Во время тестирования алгоритма наблюдались редкие случаи, приводившие к подобному состоянию журналов. Такое может произойти, например, если в процессе работы кластера из-за внутренних задержек изменения состояния некоторое время допускалась запись на два изолированных по сети друг от друга узла, причем на тот узел, который в схеме алгоритма будет соответствовать A, данных было записано меньше.

Для полного избавления от подобных эффектов необходимы более вычислительно затратные решения, например, последовательное чтение и сравнение записей журнала или попытки вычисления какого-либо рода хеш-функций для сравнения содержимого, с учетом разного количества записей на узлах. В данной работе такие решения не рассматриваются в силу достаточной доли сценариев, в которых алгоритм достигает поставленной цели.

Результаты

Описанный выше подход был реализован в виде набора модификаций ядра PostgreSQL 16.1 и расширения для той же версии. Модификации и расширение написаны на языке C. Для создания нагрузки при тестировании производительности использовалась нагрузочная утилита pgbench со стандартным TPC-C-like сценарием и соответствующим масштабированием.

Эталонная реализация алгоритма состоит из двух частей: расширения для передачи heartbeat-сообщений и принятия решений; и модификаций ядра СУБД для управления процессом репликации.

В расширении, помимо установки соединений и передачи сообщений, извлекаются и поддерживаются значения номеров последних записей журнала и файла истории последней линии времени. Состав кластера задается пользователем в конфигурационном файле. При запуске СУБД расширение немедленно, до запуска логики восстановления после сбоя, стартует один процесс-обработчик, периодически отправляющий сообщения.

Когда применение всех доступных записей журнала завершено и необходимая информация получена с узла-источника, в расширении принимается решение о дальнейшем поведении репликации согласно матрице принятия решения.

Расширение и ядерные изменения взаимодействуют посредством выделяемой в PostgreSQL для вспомогательных процессов разделяемой памяти.

В случае если решением, которое принял алгоритм, оказалось «нельзя / синхронизация», поведение зависит от параметров конфигурации. Выполнение синхронизации при помощи pg_rewind вызывает перезагрузку СУБД и автоматически производит синхронизацию, если пользователь установил соответствующий параметр конфигурации. Если этот параметр отсутствует, запуск репликации блокируется.

Для тестирования использовались два способа создания кластеров, с помощью кластерных утилит patroni версии 4.0.5 [16] и Postgres Pro BiHA в составе PostgresPro Enterprise 16.1 [17].

Все три варианта подразумевают кластер из трех узлов, с одним ведущим и двумя ведомыми, получающими данные посредством асинхронной физической репликации. Ведущий узел работает в обычном режиме и доступен клиенту на чтение и запись. При выходе ведущего узла из строя или изоляции относительно ведомых один из ведомых становится новым ведущим узлом. Все узлы запускаются внутри виртуальных машин на одном физическом сервере.

Критерий оценки эффективности алгоритма – способность узла, бывшего ведущим, вернуться в кластер и продолжить работу в качестве резервного узла, имея корректный набор данных и корректно применяя новые изменения с нового ведущего. В качестве типовых сце-

нариев, когда ожидается подобное поведение, использовалось переключение ведущего узла (switchover [18]), и возникновение split-brain ситуаций вследствие сетевого разделения кластера. Корректность набора данных проверялась посредством попарного сравнения записей в тестовой таблице между введенным узлом и новым ведущим узлом.

Сетевое разделение считается частой причиной сбоев в распределенных системах [19]. В работе сетевое разделение эмулировалось посредством изменения правил *iptables* на узлах для отклонения всех входящих и исходящих пакетов и через настраиваемый временной интервал – отмены таких правил. Такой подход позволяет блокировать любое сетевое взаимодействие между выбранными узлами кластера, эффективно заставляя их считать друг друга вышедшими из строя. После разделения выполнялись записи на оба ведущих узла – на старый, пока узел еще не запретил запись, и на новый, после того как он был избран кластерным ПО.

Переключение ведущего узла обычно определяется как контролируемое аварийное переключение ведущего узла с балансировкой нагрузки. Если кластер СУБД построен с использованием асинхронной репликации, то отказы во временном интервале между принятием решения о переключении и моментом, когда узел-кандидат на роль нового ведущего получит все необходимые данные посредством репликации, могут привести к расхождению данных старого и нового ведущего. Например, если текущий ведущий становится недоступным во время процесса переключения, а кандидат не получил все необходимые данные, то кандидат будет вынужден стать новым ведущим с неполным комплектом данных.

При тестировании во время переключений эмулировались внешние сбои: останов узла или сетевое разделение. Для эмуляции замедления из-за пиковой нагрузки на случайных временных интервалах запускались дополнительные требовательные к процессорному времени процессы и/или урезались доступные узлу ресурсы. Если выполнялись разделение или останов узла, после завершения переключения восстанавливалось исходное состояние: правила iptables удалены, все узлы запущены.

Каждый из сценариев воспроизводился по 2000 раз для кластера с загруженным расширением, реализующим алгоритм, и без него. Для сценария переключения лидера случайным образом, но не менее одного во время каждого переключения генерировались события выключения или сетевого разделения узлов кластера.

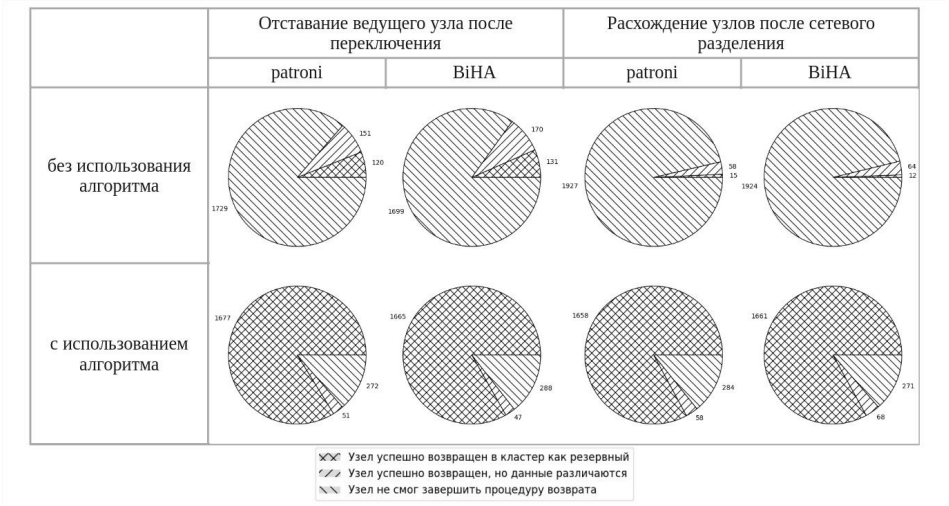

Рис. 6. Число успешных возвратов бывшего ведущего узла в кластер как резервного
Fig. 6. Amount of successful former primary node's readditions as standby

Результаты приведены на рис. 6. Стоит отметить, что без использования алгоритма вероятность успешного ввода в строй после эмулируемого сценария невысока. Использование

предложенных проверок повышает вероятность успешного ввода до приемлемых значений. Остальные случаи относятся к краевым и требуют дополнительного сравнения содержимого журналов, что описано в разделе о матрице решений.

Использование реализованного в расширении алгоритма незначительно увеличивает время восстановления после сбоя. Основные факторы – отказ от записи реплицированных одновременно с восстановлением и ожидание получения информации от узла-источника. В работе не приводятся исчерпывающие измерения, поскольку влияние данных факторов зависит от объемов журнала при восстановлении, сетевой инфраструктуры, используемого внешнего ПО для обслуживания кластера (в частности, использование такого ПО тоже привносит задержки из-за обмена информацией о состоянии узлов). На тестовых стендах разница среднего времени восстановления для сценариев без и с использованием расширения не превысила 3 % от абсолютного значения.

Заключение

В работе предложен и реализован алгоритм проверки предусловий корректности запуска репликации с минимальными накладными расходами. На типовых для отказоустойчивого кластера сценариях алгоритм обнаружил большинство некорректных ситуаций и предотвратил сбой узла из-за запуска репликации. Предлагаемая алгоритмом автоматическая синхронизация узлов при помощи утилиты pg_rewind позволила запустить репликацию на узел корректным образом и вернуть узел в кластер без вмешательства администратора.

Таким образом, в данных сценариях при реальном сбое использование алгоритма позволит ввести узел в состав кластера без необходимости создания нового, что значительно сокращает время ввода узла в кластер.

В работе также описаны не детектируемые алгоритмом пограничные сценарии, которые требуют разработки дополнительных, более требовательных к ресурсам проверок перед запуском репликации.

Для автоматической синхронизации узлов в работе используется утилита pg_rewind. Из-за архитектурных особенностей утилиты для завершения синхронизации требуется априорный запуск репликации при запуске узла. Для этого требуется единоразовое отключение логики предлагаемого алгоритма. Проектирование более подходящего способа синхронизации узлов, между которыми нельзя запустить репликацию, может быть направлением дальнейшего исследования.

Список литературы

1. **Thomas S. M.** PG Phriday: Redefining Postgres High Availability // BonesMoses.org: сайт. 2024. URL: https://bonesmoses.org/2024/pg-phriday-redefining-postgres-high-availability/
2. **Kassema J. J.** Disaster Recovery Plan for Business Continuity: Case Study in a Business Sector. In: SSRN. 2016. DOI: 10.2139/ssrn.2796601
3. **Stonebraker M., Rowe L. A.** The design of POSTGRES. In: Proceedings of the 1986 ACM SIGMOD international conference on Management of data (SIGMOD '86) (June 1986). Association for Computing Machinery, New York, NY, USA, 1986. P. 340–355. DOI 10.1145/16856.16888
4. **Cecchet E., Candea G., Ailamaki A.** Middleware-based database replication: the gaps between theory and practice // Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD '08). Association for Computing Machinery, New York, NY, USA, 2008. P. 739–752. DOI 10.1145/1376616.1376691
5. **Stonebraker M., Rowe L., Hirohama M.** The Implementation Of Postgres // Knowledge and Data Engineering, IEEE Transactions on. 2. 1990. P. 125–142. DOI 10.1109/69.50912
6. **Wieck J.** Slony-I. A replication system for PostgreSQL // Slony-I. URL: https://slony.info/images/Slony-I-concept.pdf

7. PostgreSQL: Documentation 9.0: Release 9.0 // PostgreSQL: Documentation. URL: https://www.postgresql.org/docs/9.0/release-9-0.html
8. **Linnakangas H.** Understanding PostgreSQL timelines // FOSDEM 2013. URL: https://wiki.postgresql.org/images/e/e5/FOSDEM2013-Timelines.pdf
9. **Davidson S. B., Garcia-Molina H.; Skeen D.** Consistency In A Partitioned Network: A Survey // ACM Computing Surveys. 1985. Vol. 17, iss. 3. P. 341–370. DOI 10.1145/5505.5508
10. **Панченко И.** PostgreSQL: вчера, сегодня, завтра // Открытые системы. СУБД. 2015. № 3. С. 34–37. URL: https://www.osp.ru/os/2015/03/13046900
11. PostgreSQL: Documentation 17.0: pg_rewind // PostgreSQL: Documentation. URL: https://www.postgresql.org/docs/17/app-pgrewind.html
12. **Härder T., Sauer C., Graefe G. et al.** Instant recovery with write-ahead logging // Datenbank Spektrum. 2015. Vol. 15. P. 235–239. DOI 10.1007/s13222-015-0204-3.
13. **Bárbaro P., Pedroso M.** High Availability and Load Balancing for Postgresql Databases: Designing and Implementing. In: International Journal of Database Management Systems. 2016, vol. 8, pp. 27–34. DOI 10.5121/ijdms.2016.8603
14. **Md. Anower H., Md. Imrul H., Dr. MD Rashedul I., Nadeem A.** A Novel Recovery Process in Timelagged Server using Point in Time Recovery (PITR). In: 24th International Conference on Computer and Information Technology (ICCIT). 2021. DOI: 10.1109/ICCIT54785.2021.9689808.
15. **Kim H., Yeom H. Y, Son Y.** An Efficient Database Backup and Recovery Scheme using Write-Ahead Logging // 2020 IEEE 13th International Conference on Cloud Computing (CLOUD). 2020. P. 405–413. DOI 10.1109/CLOUD49709.2020.00062
16. Introduction – patroni 3.3 documentation // Patroni documentation. https://patroni.readthedocs.io/en/rel_3_3/
17. Postgres Pro Enterprise: Документация: 16: F.8: biha – встроенный отказоустойчивый кластер // Документация PostgreSQL и Postgres Pro: компания Postgres Professional. URL: https://postgrespro.ru/docs/enterprise/16/biha
18. **Meng-Lai Y.** Assessing availability impact caused by switchover in database failover // 2009 Annual Reliability and Maintainability Symposium. Fort Worth, TX, USA. 2009. P. 401–406. DOI 10.1109/RAMS.2009.4914710.
19. **Coan B. A., & Oki B. M., Kolodner E. K.** Limitations on Database Availability when Networks Partition // PODC ‘86: Proceedings of the fifth annual ACM symposium on Principles of distributed computing. 1986. P. 187–194. DOI: 10.1145/10590.10606.

References

1. **Thomas S. M.** PG Phriday: Redefining Postgres High Availability. In: BonesMoses.org: сайт. 2024. URL: https://bonesmoses.org/2024/pg-phriday-redefining-postgres-high-availability/
2. **Kassema J. J.** Disaster Recovery Plan for Business Continuity: Case Study in a Business Sector. In: SSRN, 2016, DOI: 10.2139/ssrn.2796601
3. **Stonebraker M., Rowe L. A.** The design of POSTGRES. In: Proceedings of the 1986 ACM SIGMOD international conference on Management of data (SIGMOD ‘86) (June 1986). Association for Computing Machinery, New York, NY, USA, 1986, pp. 340–355. DOI 10.1145/16856.16888
4. **Cecchet E., Candea G., Ailamaki A.** Middleware-based database replication: the gaps between theory and practice. In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD ‘08)*. Association for Computing Machinery, New York, NY, USA, 2008, pp. 739–752. DOI 10.1145/1376616.1376691
5. **Stonebraker M., Rowe L., Hirohama M.** The Implementation Of Postgres. In: *Knowledge and Data Engineering, IEEE Transactions on. 2.* 1990, pp. 125–142. DOI 10.1109/69.50912

6. **Wieck J.** Slony-I. A replication system for PostgreSQL. In: Slony-I. URL: https://slony.info/images/Slony-I-concept.pdf
7. PostgreSQL: Documentation 9.0: Release 9.0. In: PostgreSQL: Documentation. URL: https://www.postgresql.org/docs/9.0/release-9-0.html
8. **Linnakangas H.** Understanding PostgreSQL timelines. In: FOSDEM 2013. URL: https://wiki.postgresql.org/images/e/e5/FOSDEM2013-Timelines.pdf
9. **Davidson S. B., Garcia-Molina H.; Skeen D.** Consistency In A Partitioned Network: A Survey. In: *ACM Computing Surveys*, 1985, vol. 17, iss. 3, pp. 341–370. DOI 10.1145/5505.5508
10. **Panchenko I.** PostgreSQL: yesterday, today, tomorrow. In: Open systems. DBMS, 2015, no. 3, pp. 34–37. URL: https://www.osp.ru/os/2015/03/13046900
11. PostgreSQL: Documentation 17.0: pg_rewind. In: PostgreSQL: Documentation. URL: https://www.postgresql.org/docs/17/app-pgrewind.html
12. **Härder, T., Sauer, C., Graefe, G. et al.** Instant recovery with write-ahead logging. *Datenbank Spektrum* 15. 2015, pp. 235–239. DOI 10.1007/s13222-015-0204-3.
13. **Bárbaro P., Pedroso M.** High Availability and Load Balancing for Postgresql Databases: Designing and Implementing. *International Journal of Database Management Systems*, 2016, vol. 8, pp. 27–34. DOI 10.5121/ijdms.2016.8603
14. **Md. Anower H., Md. Imrul H., Dr. MD Rashedul I., Nadeem A.** A Novel Recovery Process in Timelagged Server using Point in Time Recovery (PITR). In: *24th International Conference on Computer and Information Technology (ICCIT)*. 2021. DOI 10.1109/ICCIT54785.2021.9689808.
15. **Kim H., Yeom H. Y, Son Y.** An Efficient Database Backup and Recovery Scheme using Write-Ahead Logging. In: *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, 2020, pp. 405–413, DOI: 10.1109/CLOUD49709.2020.00062
16. Introduction – patroni 3.3 documentation. In: patroni documentation. https://patroni.readthedocs.io/en/rel_3_3/
17. Postgres Pro Enterprise: Documentation: 16: F.8: biha – built-in high-availability cluster // Documentation PostgreSQL и Postgres Pro: Postgres Professional: site. URL: https://postgrespro.ru/docs/enterprise/16/biha
18. **Meng-Lai Y.** Assessing availability impact caused by switchover in database failover. In: *2009 Annual Reliability and Maintainability Symposium*. Fort Worth, TX, USA, 2009, pp. 401–406. DOI: 10.1109/RAMS.2009.4914710.
19. **Coan B. A., & Oki B. M., Kolodner E. K.** Limitations on Database Availability when Networks Partition. In: *PODC '86: Proceedings of the fifth annual ACM symposium on Principles of distributed computing*. 1986, pp. 187–194. DOI: 10.1145/10590.10606.

Информация об авторах

Рудометов Андрей Сергеевич, студент магистратуры; ассистент

Рутман Михаил Валерьевич, доцент

Information about the Authors

Andrey S. Rudometov, Master's Student

Mikhail V. Rutman, Associate Professor

Статья поступила в редакцию 10.03.2025;
одобрена после рецензирования 07.04.2025; принята к публикации 07.04.2025

The article was submitted 10.03.2025;
approved after reviewing 07.04.2025; accepted for publication 07.04.2025