Ай-ТИ Фреш

ТЕХНИЧЕСКИЙ РАЗБОР

PostgreSQL 18: почему 40001 возвращается после SAVEPOINT

Наш разбор serialization_failure в REPEATABLE READ/SERIALIZABLE и правильный retry-цикл

Июль 2026

itfresh.ru · ИТ-аутсорсинг для юридических лиц

О ЧЁМ ЭТОТ РАЗБОР

Суть проблемы

Клиент ловит 40001, делает ROLLBACK TO SAVEPOINT и повторяет UPDATE — ошибка возвращается. Откат к savepoint не трогает снимок данных: он взят на первом не-транзакционном операторе и живёт до COMMIT/ROLLBACK, предикатные SIRead-блокировки тоже сохраняются. Повтор читает те же строки и принимает то же решение, конфликт воспроизводится один в один — операция виснет в цикле.

Почему это важно бизнесу

- 40001 читают как «просто повторить запрос»: без рестарта транзакция виснет и держит закрытие смены или накладной
- Ретрай через SAVEPOINT работает на старом снимке и может закоммитить поверх чужой правки — расхождение остатков
- Пустые повторы копят SIRead-блокировки и грузят CPU при конкурентной записи в одну строку остатков
- Кассир или пользователь 1С видит зависшую оплату, пока backend чинит обречённую транзакцию
- Замаскированный конфликт искажает отчёт по остаткам, и расхождение всплывает уже на инвентаризации

ЧТО МЫ ЗАКЛАДЫВАЕМ

Ключевые параметры реализации

40001

SQLSTATE serialization_failure, класс 40: требуется рестарт всей транзакции

Appendix A. Error Codes, class 40

64

max_pred_locks_per_transaction: объектов под предикатной блокировкой на процесс

Lock Management, runtime-config-locks

2

max_pred_locks_per_page: строк на странице до эскалации SIRead на всю страницу

Lock Management, runtime-config-locks

40P01

SQLSTATE deadlock_detected: ретраим тем же циклом с нуля, что и 40001

Appendix A. Error Codes, class 40

25P02

in_failed_sql_transaction: нужен полный ROLLBACK, а не ROLLBACK TO SAVEPOINT

Appendix A. Error Codes, class 25

3-5

попыток в нашем retry_tx() на SERIALIZABLE до эскалации в очередь и алерта

регламент ITfresh retry_tx()

РЕАЛИЗАЦИЯ 1 ИЗ 3

Retry-обвязка на REPEATABLE READ и SERIALIZABLE

Что настраиваем

Бэкенды 1С-контура и сервисы на Python/Java/.NET, PostgreSQL 15-18 за пулом pgbouncer

Как мы это делаем

1. Стартуем BEGIN ISOLATION LEVEL SERIALIZABLE или REPEATABLE READ: уровень задаём до первого запроса, дефолт базы держим default_transaction_isolation = read committed
2. Ловим 40001 и 40P01 по SQLSTATE, а не по тексту: psycopg 3 — e.sqlstate, pgJDBC — getSQLState(), Npgsql — PostgresException.SqlState, pgx — pgconn.PgError.Code
3. Делаем полный ROLLBACK и новый BEGIN: только он берёт свежий снимок и обнуляет SIRead-блокировки, ROLLBACK TO SAVEPOINT этого не делает
4. В retry_tx() заворачиваем весь бизнес-блок — чтения, решения и запись, потому что решение построено на устаревших данных так же, как и сам UPDATE
5. Backoff 100-300 мс экспоненциально с джиттером, максимум 5 попыток, дальше — в очередь и алерт дежурному

РЕЗУЛЬТАТ

Полный рестарт снимает повторные 40001 на конкурентной записи в одну строку остатков: проверяли до/после на стенде 1С, повтор через savepoint давал ту же ошибку в 100% попыток

КЛЮЧЕВОЙ НЮАНС

Savepoint откатывает эффекты операторов, но не снимок и не предикатные блокировки транзакции: новый снимок берётся только на первом операторе новой транзакции

РЕАЛИЗАЦИЯ 2 ИЗ 3

Снижаем конфликтность: порядок доступа и явные блокировки

Что настраиваем

Учёт товародвижения и кассы с горячими строками остатков, PostgreSQL 17/18

Как мы это делаем

1. Приводим порядок обновления строк к единому (ORDER BY id в SELECT ... FOR UPDATE) во всех местах кода — снимаем встречную блокировку и 40P01
2. Горячие точечные UPDATE переводим на SELECT ... FOR UPDATE в READ COMMITTED: конфликт превращается в ожидание блокировки, а не в 40001
3. SERIALIZABLE оставляем только там, где есть аномалия записи-чтения; тяжёлую аналитику выносим в SERIALIZABLE READ ONLY DEFERRABLE — она не даёт и не получает 40001
4. Ставим idle_in_transaction_session_timeout = 60s, чтобы забытая сессия не держала снимок и предикатные блокировки
5. Включаем log_lock_waits, ожидание пишется после deadlock_timeout (по умолчанию 1s) — видим, кто держит строку

РЕЗУЛЬТАТ

Частота 40001 на нагруженной таблице остатков падает кратно без отказа от SERIALIZABLE там, где он реально закрывает аномалию записи-чтения

КЛЮЧЕВОЙ НЮАНС

SIReadLock никого не блокирует и не участвует в deadlock, но именно она порождает 40001 — конфликтность режем границами транзакций, а не отключением ретраев

РЕАЛИЗАЦИЯ 3 ИЗ 3

Диагностика: отличаем 40001 от 40P01 и от 23505

Что настраиваем

Дежурная эксплуатация ITfresh: разбор алертов по логам PostgreSQL 15-18 у клиентов

Как мы это делаем

1. Два текста 40001: «could not serialize access due to concurrent update» — конфликт по строке в REPEATABLE READ; «...read/write dependencies among transactions» — SSI
2. Смотрим pg_locks с mode = 'SIReadLock': гранула page или relation вместо tuple означает эскалацию по max_pred_locks_per_page или max_pred_locks_per_relation
3. Счётчик deadlocks берём из pg_stat_database, частоту serialization-ошибок — из журнала по SQLSTATE, а не по подстроке сообщения
4. 23505 и 23P01 обрабатываем отдельной веткой и повторяем только там, где ключ генерируется заново внутри транзакции
5. Помним про курсор: после аварийного завершения транзакции он переходит в состояние cannot-execute, и ROLLBACK TO SAVEPOINT его не оживляет

РЕЗУЛЬТАТ

Раздельная обработка классов 40 и 23 убирает бесконечные ретраи на честных нарушениях уникальности и сокращает время разбора инцидента до минут

КЛЮЧЕВОЙ НЮАНС

Общий catch «retry on any DB error» стирает разницу между транзакционным конфликтом и нарушением целостности данных — сценарии восстановления у них противоположные

ЧТО МЫ ВЫУЧИЛИ НА ПРАКТИКЕ

Подводные камни

- ✕ **ROLLBACK TO SAVEPOINT вместо ROLLBACK**
  Возврат к точке внутри той же транзакции: снимок и SIRead-блокировки прежние, тот же конфликт по строке воспроизводится снова
- ✕ **Повтор одного UPDATE, а не блока**
  Решение вычислено по старым чтениям того же снимка, поэтому корректен только повтор всей цепочки чтение-решение-запись с нового BEGIN
- ✕ **Ретрай без backoff и джиттера**
  Мгновенный BEGIN в цикле: конкуренты на одной горячей строке синхронно бьются в 40001 и добавляют нагрузку вместо прогресса
- ✕ **SERIALIZABLE там, где хватит FOR UPDATE**
  Оборачивание всей CRUD-логики в SERIALIZABLE множит SIRead-блокировки и даёт 40001 на несвязанных операциях
- ✕ **Смешение классов 40 и 23 в одном catch**
  Общий retry повторяет и честное нарушение 23505 — на выходе тихий дубль документа либо цикл до исчерпания попыток
- ✕ **Долгая транзакция держит снимок**
  Снимок фиксируется на первом операторе, поэтому вызов внешнего API внутри BEGIN гарантирует устаревшие данные и 40001 на записи
- ✕ **Bulk-UPDATE упирается в эскалацию**
  После max_pred_locks_per_page = 2 блокировка растёт до страницы, затем до relation — конфликтует вся таблица, а не строка
- ✕ **Повтор без нового BEGIN**
  Новый SQL уходит в уже аварийно завершённую транзакцию без ROLLBACK и BEGIN — сервер отвечает 25P02 на любой оператор

УРОВНИ РЕАЛИЗАЦИИ

Как правильно

МИНИМУМ

- Ловить 40001 и 40P01 по SQLSTATE, а не по тексту сообщения
- На конфликте делать полный ROLLBACK и новый BEGIN, а не ROLLBACK TO SAVEPOINT
- Повторять весь блок чтение-решение-запись, а не упавший оператор
- Ограничить цикл 3-5 попытками и логировать факт эскалации

НОРМАЛЬНО

- Backoff 100-300 мс экспоненциально с джиттером между попытками
- Задать idle_in_transaction_session_timeout = 60s на боевых базах
- Развести обработку класса 40 (рестарт) и класса 23 (данные)
- Включить log_lock_waits и следить за pg_stat_database.deadlocks

ХОРОШО

- Горячие строки закрывать SELECT ... FOR UPDATE в READ COMMITTED
- Аналитику выносить в SERIALIZABLE READ ONLY DEFERRABLE
- Единый порядок обновления строк (ORDER BY id) во всём коде
- Алерт на рост SIReadLock гранулы page/relation в pg_locks

ПРОВЕРЬТЕ У СЕБЯ

Чек-лист самопроверки

- [ ] Ретрай делает полный ROLLBACK и новый BEGIN, а не ROLLBACK TO SAVEPOINT?
- [ ] Повторяется вся бизнес-функция целиком, включая чтения и решения?
- [ ] Ошибки ловятся по SQLSTATE (40001/40P01), а не по тексту сообщения?
- [ ] Есть лимит попыток и эскалация в очередь или алерт при исчерпании?
- [ ] Между попытками стоит экспоненциальный backoff с джиттером?
- [ ] Класс 40 и класс 23 (23505/23P01) обработаны разными ветками?
- [ ] Задан idle_in_transaction_session_timeout на боевых базах?
- [ ] Известны max_pred_locks_per_transaction и запас на bulk-операции?
- [ ] Включён log_lock_waits, снимается pg_stat_database.deadlocks?
- [ ] В pg_locks нет SIReadLock с гранулой page или relation на горячих таблицах?

Если хотя бы на два вопроса ответ «нет» или «не знаю» — тема требует внимания.

ЧТО МЫ ДЕЛАЕМ

Как поможет ITFresh

ITFresh — ИТ-аутсорсинг для юридических лиц до 50 рабочих мест в Москве и области. 15+ лет практики, собственная инфраструктура в дата-центре МТС (8 серверов Dell Xeon Platinum).

- Аудит уровней изоляции и границ транзакций в 1С-контуре и веб-системах
- Внедряем retry_tx() с backoff и лимитом попыток: Python, Java, .NET, PHP
- Настраиваем pg_locks, pg_stat_database и алерты на serialization-конфликты
- Пересобираем порядок доступа к горячим строкам без правки бизнес-логики
- Сопровождаем кластер PostgreSQL: версии, predicate locks, таймауты, логи

15+
лет в ИТ-поддержке

50
рабочих мест — наш профиль

МТС
дата-центр, Москва

КОНТАКТЫ

Обсудить вашу задачу

Сайт	**itfresh.ru**
Телефон	**+7 903 729-62-41**
Telegram	**@ITfresh_Boss**

Бесплатно посмотрим вашу инфраструктуру по этому чек-листу и скажем, где тонко — без обязательств.

itfresh.ru

ДОКУМЕНТАЦИЯ И НАШ СТЕК

Техническая база

01 13.5. Serialization Failure Handling (postgresql.org — 18)

02 13.2. Transaction Isolation (Repeatable Read, Serializable) (postgresql.org — 18)

03 Appendix A. PostgreSQL Error Codes, classes 25 и 40 (postgresql.org — 18)

04 Server Configuration: Lock Management (max_pred_locks_*) (postgresql.org — 18)

05 SQL Commands: SAVEPOINT, ROLLBACK TO SAVEPOINT (postgresql.org — 18)

06 Monitoring: pg_locks, pg_stat_database (postgresql.org — 18)

07 Transactions: SerializationFailure, Error.sqlstate (psycopg.org — 3.2)

08 PSQLState.SERIALIZATION_FAILURE, getSQLState() (jdbc.postgresql.org — 42.7)

09 Шаблон retry_tx() и регламент ретраев ITfresh (itfresh.ru — 2026)

Основано на официальной документации продуктов и нашей практике внедрения.