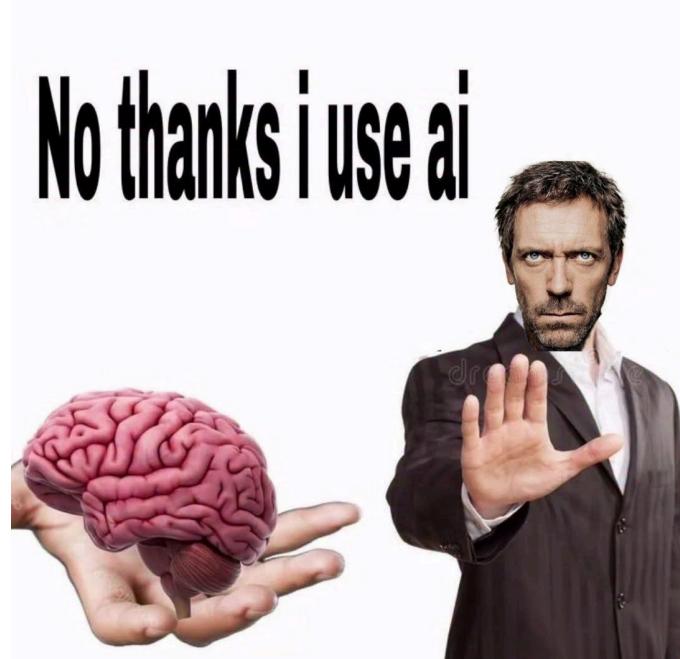


Эссе "Почему BIGSERIAL почти всегда лучше UUID для первичных ключей, даже в 2025 году"

Выполнил: Васюков Александр Владимирович, БПИ-235



Вступление

Обсуждая с AI тему "Почему BIGSERIAL почти всегда лучше UUID для первичных ключей, даже в 2025 году?", я нашёл ответы на следующие вопросы:

- Какие есть преимущества у UUID?
- Есть ли обратные стороны у этих преимуществ?
- В чём главный недостаток UUID?
- Подтверждаются ли гипотезы о недостатках UUID на практике?

Первым промптом были получены сильные аргументы "за" использование UUID как первичных ключей. После этого был выбран наиболее весомый аргумент и приведены контраргументы с альтернативами. Третьим промптом были сгенерированы DDL, сценарии нагружочного тестирования для сравнения таблицы с UUID и таблицы с BIGSERIAL. На основе ответов нейросети получилось данное эссе. В конце приведены логи взаимодействия с AI.

Обзор сильных сторон UUID

AI выделил несколько сильных сторон UUID.

- **Глобальная уникальность:**

UUID даёт уникальный идентификатор, который практически никогда не повторяется, даже если записи создаются в разных местах. Это удобно, когда в системе много независимых узлов и нельзя централизованно выдавать номера.

- **Можемость генерировать идентификатор на клиенте:**

Клиент или сервис могут сразу присвоить объекту id, не запрашивая это у базы — это ускоряет работу и позволяет работать офлайн. Такие id можно сразу отправлять в очереди или к другим сервисам.

- **Удобство для публичных API:**

UUID хорошо подходит для ссылок в API, он фиксированной длины и не даёт понятной последовательности, поэтому постороннему человеку труднее угадать следующий ресурс. Это делает ссылки более безопасными по предсказуемости.

- **Невозможность легко угадать следующую запись:**

В отличие от простого счётчика, UUID затрудняет перечисление всех записей по порядку, что важно для

публичных URL и некоторых требований безопасности.

- **Гибкость версий:**

Существуют варианты UUID, например, полностью случайные или версии, где часть содержит метку времени — можно подобрать формат под задачу: скорость генерации, упорядоченность или приватность.

Эти преимущества делают UUID полезными во многих архитектурных сценариях. Но важно помнить, что полезность преимущества не равна безусловной пользе для всех задач. Надо смотреть на нагрузку, оборудование и требования к скорости и безопасности.

Преимущества UUID часто переоценивают

При этом для каждого преимущества можно найти контраргументы и переоцененность:

- **Производительность вставок и локальность данных:**

UUID в стандартном виде заставляет базу разбрасывать новые записи по всему индексу. Он расширяется в разных местах, в результате появляется больше операций записи на диск и снижается скорость вставок. Последовательный счётчик просто добавляет записи в конец — это быстрее и проще для системы.

- **Размер памяти:**

UUID занимает 16 байт, тогда как BIGSERIAL — 8 байт. При большом количестве записей это сказывается на заметном росте индекса и расходе памяти. Большой индекс хуже помещается в оперативной памяти, и система чаще обращается к диску.

- **Читаемость и удобство работы:**

UUID длинный и неудобный для человека: в логах, в отладке и при ручной проверке он мешает. Маленький числовой id проще читать и вводить.

- **Интеграция с legacy-кодом:**

Многие старые системы, отчёты и инструменты работают с небольшими числовыми id. Интеграция UUID требует дополнительной логики преобразования и может усложнить поддержку.

Альтернативные решения

ULID/KSUID и Snowflake дают распределённую генерацию и одновременно обеспечивают упорядоченность по времени — это уменьшает разброс записей в индексе. Выделение диапазонов чисел для каждого узла даёт преимущества локальной генерации, сохраняя компактность BIGSERIAL. Гибрид с BIGSERIAL и отдельным public_id (UUID) часто оказывается самым практичным: внутренняя работа — с компактным числом, внешний API — с UUID.

Главный недостаток

Главная проблема UUID как первичного ключа — это разброс новых записей по структуре индекса. Легче и быстрее работать, если данные приходят в алфавитном порядке, а не в случайном, тогда вставлять новые записи можно подряд, а не искать нужное место, сдвигать и переставлять существующие. С UUID как раз и происходит разбиение страниц индекса: система вынуждена менять содержимое страниц на диске, чтобы вставить запись "в середину".

Это происходит, потому что UUID — случайные значения. Когда индекс устроен как упорядоченная структура, вставка случайного ключа почти всегда попадает не в конец, а внутрь. Чтобы освободить место, база разделяет страницу индекса на две, и это требует дополнительных записей на диск.

Последствия:

- Больше операций записи на диск
- Медленнее массовые вставки
- Более крупные индексы
- Выше нагрузка на систему при репликации и бэкапах

Как это можно исправить:

- Хранить UUID как внешний public_id, а в базе применять BIGSERIAL как первичный ключ

- Использовать упорядоченные альтернативы UUID (ULID, KSUID) — тогда новые записи чаще вставляются рядом
- Понизить fillfactor — это оставляет свободное место в страницах
- Настраивать сборщик мусора — это уменьшает частоту разбиений

Сравнение

Был также проведён эксперимент с сравнением работы UUID и BIGSERIAL в PostgreSQL. Были сгенерированы два DDL-скрипта PostgreSQL: таблица event_log_uuid с первичным ключом типа UUID через gen_random_uuid() и таблица event_log_bigserial с первичным ключом BIGINT GENERATED AS IDENTITY. Структура таблиц и вторичные индексы были одинаковыми, различался только тип первичного ключа.

Нагрузочный тест представлял собой массовую вставку 1 000 000 строк через Python-скрипты батчами по 1000 строк. Результаты показали заметную разницу: загрузка в таблицу с UUID заняла приблизительно 19.5 секунд, тогда как вставка в таблицу с BIGSERIAL — 15 секунд, то есть примерно на 25% быстрее.

Результат оказался ожидаемым. Последовательные BIGSERIAL-идентификаторы приводят к тому, что новые записи просто добавляются в конец структуры индекса, тогда как случайные UUID чаще всего требуют вставки внутри этой структуры, вызывая разделение страниц индекса, увеличение объёма журнала транзакций и ухудшение локальности данных. Даже на сравнительно небольшом объёме данных это даёт измеримый проигрыш UUID по производительности.

Практические рекомендации

- Для внутренних систем с высокой нагрузкой и централизованной генерацией id лучше использовать BIGSERIAL. Это компактнее, быстрее по вставке, даёт меньшие индексы и простую интеграцию со старыми системами.
- Если нужен внешний публичный идентификатор, лучше добавить отдельный public_id типа UUID, но оставить id как BIGSERIAL. Этот способ объединяет лучшее от двух вариантов, даёт и скорость, и удобство.
- Для распределённых систем, где нет центрального выдающего номера, можно использовать ULID/KSUID или Snowflake — они позволяют генерировать id локально и в то же время меньше ломают индексы.
- При миграции лучше добавлять UUID столбец, заполнять его, переводить API на использование public_id, а потом переходить на id как на основной ключ для внутренних ссылок.

Заключение

UUID — полезный инструмент, особенно для распределённых сценариев и публичных API. Но в роли первичного ключа для таблиц с большой интенсивностью вставок он часто создаёт критические проблемы: большие индексы, больше работы с диском и более сложную настройку. Для большинства практических задач BIGSERIAL остаётся более pragматичным и эффективным решением. Если нужны свойства UUID — лучше использовать их целенаправленно как public_id или выбирать альтернативы, которые сохраняют упорядоченность и компактность.

Логи

Обсуждение велось с нейросетью ChatGPT-5 от OpenAI.

Промпт 1

Ты – технический эксперт по СУБД и архитектурам данных.
Наша цель – подготовить диалог для митапа "Почему BIGSERIAL почти всегда лучше UUID для первичных ключей, даже в 2025 году".
Начнём с того, что ты перечислишь **всё** сильное "за" использование UUID в качестве первичных ключей.

Инструкции к ответу:

1. Выведи короткую шапку: "Промпт 1 – аргументы за UUID".
2. Дай структурированный список аргументов. Для каждого аргумента приведи:
 - Заголовок (1-3 слова).
 - Краткое предложение (1-2 строки), объясняющее суть.
 - Техническое обоснование (3-6 предложений) – почему это преимущество, в каких сценариях оно действительно важно.
 - Примеры практического применения (1-2 примера: микросервисы, распределённые системы, offline-синхронизация, public APIs и т.п.).
 - Потенциальные контр-ограничения (1-2 предложения), т.е. когда это преимущество несущественно.
3. Включи отдельный блок "Самые сильные аргументы" – отсортируй 3 лучших аргумента по силе влияния (1-3) и поясни почему именно они самые весомые.
4. Объём ответа: примерно 8-15 пунктов аргументов; общий объём ~500-900 слов.
5. В конце добавь короткую заметку: "Добавь этот вывод в Лог взаимодействия как запись №1" (AI должен вернуть этот текст вместе с ответом, чтобы лог собирался).

Форматирование: используй заголовки и нумерованные/маркированные списки. Приводи SQL-имена функций и UUID-версии (v1, v4, v6/v7 если применимо) где это релевантно.

Ответ 1

Промпт 1 – аргументы за UUID

Ниже — полный, структурированный набор «за» использование UUID в качестве первичных ключей. Для каждого пункта: заголовок, суть, техническое обоснование, примеры применения и краткое замечание, когда это преимущество слабо.

1. **Глобальная уникальность**

Коротко: 128-битный UUID обеспечивает практически глобальную уникальность без централизованного координирования.

Техническое обоснование: UUID — 128 бит, генерируются по разным алгоритмам (v1 — с timestamp/MAC, v4 — случайный, v7 — таймстемп+рандом и т.п.), что делает вероятность коллизии практически нулевой в реальных системах; это освобождает от необходимости синхронизировать счётчик между узлами. Такие свойства стандартизованы в спецификах RFC/drafts. ([Wikipedia][1])

Примеры: распределённые микросервисы, онлайн-мобильные клиенты, cross-region репликация.

Ограничения: если вся система локальна и один экземпляр БД — преимущество минимально.

2. **Генерация на клиенте (client-side IDs)**

Коротко: ключи можно создавать на клиенте без round-trip к серверу.

Техническое обоснование: функции генерации (на языке/в библиотеке) позволяют выдавать UUID до INSERT — упрощает батчи, кэширование и offline-режим; нет блокировок или дополнительных транзакций для получения id. В Postgres есть встроенный `uuid`-тип и функции генерации (`gen_random_uuid()`, `uid_generate_v4()` через расширения). ([PostgreSQL][2])

Примеры: мобильные приложения, event sourcing, отправка локальных транзакций в очередь.

Ограничения: если нужен компактный автоинкрементный номер для читаемости — не подходит.

3. **Удобство объединения данных из разных источников**

Коротко: с UUID легко объединять записи из разных баз/реплик.

Техническое обоснование: при слиянии данных из нескольких инстансов риск конфликтов по ключам минимален, не требуется ремаппинг id; это ускоряет ETL/слияние бэкап-дампов и миграции.

Примеры: multi-tenant базы, импорт/экспорт между окружениями.

Ограничения: при очень большом объёме данных накладные расходы на хранение/индексы растут.

4. **Без утечки внутренней информации**

Коротко: UUID не раскрывает объёмы или порядок вставок (особенно v4).

Техническое обоснование: последовательные BIGSERIALs прямо указывают на количество записей/те

мп роста; случайные UUIDv4 скрывают эту информацию, что полезно для публичных API и безопасности. (Впрочем, версии с таймстемпом, как v1 или v7, частично раскрывают время.) ([Wikipedia][1])

Примеры: публичные REST API, выдача ссылок/токенов.

Ограничения: UUIDv1 может утечку MAC/времени; выбор версии важен.

5. **Идентификаторы устойчивы к миграциям/рефакторингу**

Коротко: UUID остаётся валидным при переносе сущностей между сервисами.

Техническое обоснование: при реструктуризации данных или переносе сущностей между доменами нет риска перекрытия id; это упрощает масштабные рефакторинги и интеграции.

Примеры: разделение монолита на микросервисы, перенос таблиц между базами.

Ограничения: требует согласованной схемы типов и индексов.

6. **Поддержка в PostgreSQL и экосистеме**

Коротко: `uuid` — нативный тип, есть функции и расширения.

Техническое обоснование: PostgreSQL поддерживает тип `uuid`, функции `gen_random_uuid()` (pgcrypt o/ core) и `uuid-ossp` (`uuid_generate_v1/4`), индексацию и операции сравнения; экосистема ORM/SDK по всему поддерживает UUID. ([PostgreSQL][2])

Примеры: использование `uuid` в DDL, дефолтные значения `DEFAULT gen_random_uuid()` при вставке.

Ограничения: дополнительные расширения могут быть не включены по умолчанию в некоторые дистрибутивы.

7. **Отсутствие центрального узла — высокая доступность**

Коротко: генерация UUID не требует сервиса ID-allocation.

Техническое обоснование: можно избежать единой точки отказа, так как любой узел генерирует уникальные id локально; это упрощает масштабирование и снижает задержки на выделение id.

Примеры: горизонтально масштабируемые сервисы, edge-узлы.

Ограничения: при желании контролировать последовательность записей (clustering) придётся выбирать подходящую версию.

8. **Совместимость с распределёнными трейсинг/логированием**

Коротко: UUID удобно использовать как correlation id.

Техническое обоснование: единный 128-битный идентификатор используется и в логах, и в трейсах, что упрощает трассировку запросов через границы сервисов.

Примеры: распределённый tracing (OpenTelemetry), централизованная аналитика.

Ограничения: если system observability зависит от упорядоченного счётчика — не даёт преимущества.

9. **Гибкость версии UUID (v1/v4/v6/v7 и др.)**

Коротко: можно подобрать версию под задачу — случайность, тайм-упорядоченность или смешанный режим.

Техническое обоснование: v4 — полностью случайные; v1 — timestamp+node (может раскрывать MAC); новее v6/v7 предлагают терм-упорядоченные варианты, которые уменьшают фрагментацию индексов, сохраняя распределённость. RFC/черновики описывают эти варианты. ([IETF Datatracker][3])

Примеры: v4 для приватности API, v7 для частично упорядоченных ключей в высоконагруженных БД.

Ограничения: выбор версии влияет на поведение индексов и безопасность.

10. **Удобство в API и внешних ссылках**

Коротко: UUID-id красиво и безопасно выглядит в URL и публичных схемах.

Техническое обоснование: человекочитаемый, фиксированной длины, сложно угадать; подходит для ссылок в письмах, токенов доступа и т.п.

Примеры: публичные ссылки на объект `/items/3f9a...-...`, shareable tokens.

Ограничения: длинная строка, хуже для UX (короткие числовые id читаются легче).

Самые сильные аргументы (топ-3)

1. **Глобальная уникальность** — ключевое: отсутствие необходимости централизованного счётчика к ардинально упрощает архитектуру распределённых систем и offline-клиентов. Это фундаментальное преимущество UUID. ([Wikipedia][1])

2. **Генерация на клиенте / отсутствие round-trip** — влияние на производительность и UX (меньше за

держек, батчи, offline) делает это один из самых практических плюсов. ([PostgreSQL][2])

3. **Удобство слияния/миграций** — при реальных операциях с несколькими источниками данных (ETL, репликация, multi-region) экономия времени и риска конфликтов часто важнее накладных расходов UUID. ([Neon][4])

Добавь этот вывод в Лог взаимодействия как запись №1

Если нужно — подготовлю противоположный список «за BIGSERIAL» и сравнение по таблице (память, индекс, латентность, операции вставки) с конкретными бенчмарками (включая ссылки и цифры). Смело говори, если хочешь сразу таблицу-сравнение.

[1]: https://en.wikipedia.org/wiki/Universally_unique_identifier?utm_source=chatgpt.com "Universally unique identifier"

[2]: https://www.postgresql.org/docs/current/functions-uuid.html?utm_source=chatgpt.com "Documentation: 18: 9.14. UUID Functions"

[3]: https://datatracker.ietf.org/doc/html/draft-ietf-uuidrev-rfc4122bis-14?utm_source=chatgpt.com "draft-ietf-uuidrev-rfc4122bis-14"

[4]: https://neon.com/postgresql/postgresql-tutorial/postgresql-uuid?utm_source=chatgpt.com "PostgreSQL UUID Data Type"

Промпт 2

Продолжая предыдущую сессию: отталкиваясь от твоего ответа на "Промпт 1" выбери **самый сильный** аргумент "за" UUID (если считаешь несколько равными – выбери тот, который чаще всего приводят на митапах и статьях).

Назовём его "Опорный аргумент".

Задача: построй исчерпывающую контраргументацию к этому опорному аргументу и предложи **альтернативные решения**, которые дают те же или очень похожие преимущества **без** использования UUID как PK. Для каждой альтернативы укажи плюсы/минусы и степень сложности внедрения.

Инструкции к ответу:

1. Шапка: "Промпт 2 – контраргументы к: «<вставь сюда выбранный опорный аргумент>»".

2. Краткое (1-2 предложения) формулирование опорного аргумента.

3. Набор контраргументов (4-7 пунктов). Для каждого:

- Короткая формулировка контраргумента.
- Техническое обоснование (2-4 предложения).
- Примеры, где это проявляется на практике.

4. Альтернативные решения (минимум 4), каждое подробно:

- Название (например: "NATed ID service + tenancy", "ULID/KSUID/ordered UUID", "snowflake-style IDs", "composite keys (service_id + serial)", "UUID только как business-id, а PK – BIGSERIAL").

- Как работает (1-3 предложения).

- Какие свойства UUID оно воспроизводит (глобальная уникальность, предсказуемость, сортируемость и т.д.).

- Краткая оценка сложности внедрения (низкая/средняя/высокая).

- Возможные недостатки по сравнению с чистым UUID.

5. Включи практическую рекомендацию: "Для каких архитектур/случаев использовать какую альтернативу" (таблица 3–4 строк).

6. Объём ~500–900 слов.

7. В конце: "Добавь этот вывод в Лог взаимодействия как запись №2".

Требование к тону: строгая, техническая аргументация; при необходимости используй примеры конфигураций/портов/форматов ID.

Ответ 2

Промпт 2 – контраргументы к: «Глобальная уникальность (UUID обеспечивает уникальность по всей распределённой системе без централизованного координирования)»

Кратко опорный аргумент: UUID (128-бит) позволяет генерировать уникальные идентификаторы локально на любом узле/клиенте без обращения к централизованному сервису, что убирает точку координации и риск конфликтов при слияниях данных.

Контраргументы (4–7)

1. **Проблемы производительности индексации**

Технически: случайные 128-битные ключи (v4) приводят к сильной фрагментации B-tree индексов и повышенным I/O при вставках; это увеличивает latency вставки и размер индекса. На больших объемах разница в throughput между последовательным `BIGSERIAL` и случайным `UUID` может быть существенной.

Практика: OLTP-базы с миллионами вставок/сек наблюдают рост IO и частые page splits.

2. **Оверхед хранения и передачи**

Технически: UUID занимает 16 байт (в бинаре) vs 8 байт у `BIGINT`; в представлении как строки — ~36 символов. Это увеличивает объем индексов, кешей и сетевой трафик (JSON API).

Практика: широкие таблицы, репликация и аналитические выгрузки становятся тяжелее.

3. **Лишняя децентрализация при простых требованиях**

Технически: если архитектура — монолит/одна БД или централизованный сервис, глобальная генерация ненужна; дополнительная стоимость UUID не оправдана.

Практика: внутренние корпоративные БД, где все операции проходят через централизованный API.

4. **Проблемы читаемости и UX**

Технически: длинные UUID неудобны в логах, консоли, URL (копирование/проверка вручную) — повышают вероятность опечаток и ошибок.

Практика: ручная отладка или пользователи, вводящие id вручную.

5. **Скрытые утечки информации у некоторых версий**

Технически: v1 содержит timestamp и node (MAC) — может утечь время/идентификатор узла; выбор версии критичен.

Практика: системы, где конфиденциальность метаданных важна, требуют v4 или v7 с осторожностью.

6. **Конфликты бизнес-логики и внешних систем**

Технически: при интеграции с внешними сервисами, ожидающими маленькие целочисленные id (legacy), использование UUID требует дополнительного мэппинга и усложняет API.

Практика: интеграция с ERP/1C/старым ПО.

Альтернативные решения (минимум 4)

1. **Snowflake-style (Twitter) IDs**

Как работает: 64-битный числовой ID = timestamp (41b) + machine id (10b) + sequence (12b); генерируется локально, уникален в кластере.

Воспроизводит: глобальную уникальность, упорядоченность по времени (частично), компактность (8 байт).

Сложность: **средняя** (нужно выделить уникальные machine IDs, обеспечить надежность генератора).

Минусы: требует координации machine_id; при конфликте или сбое времени возможны коллизии.

2. **ULID / KSUID / Flake IDs (ordered UUID alternatives)**

Как работает: идентификатор содержит таймстемп + случайные/последовательные биты (ULID — монотонно упорядочен в Base32).

Воспроизводит: уникальность без центра, сортируемость по времени, компактная текстовая форма (UUID ~26 символов).

Сложность: **низкая** (библиотеки доступны на большинстве языков).

Минусы: чуть меньше энтропии чем UUIDv4; при неправильной генерации возможны дубликаты на разных узлах (нужна минимальная координация).

3. **NATed ID service + tenancy (Namespaced sequences)**

Как работает: каждый сервис/узел имеет свой namespace/tenant_id, внутри используют локальные последовательности (BIGSERIAL). Итоговый PK = (tenant_id << 48) | sequence.

Воспроизводит: глобальную уникальность (через namespace), компактность (`BIGINT`), локальную предсказуемость.

Сложность: **средняя/высокая** (надо управлять namespace allocation, резервировать диапазоны).

Минусы: сложнее управление при масштабировании числа tenants; возможны пробелы в последовательности.

4. **Composite key: (service_id TEXT / SMALLINT) + BIGSERIAL**

Как работает: PK — составной (service_id, local_id); уникальность достигается комбинацией.

Воспроизводит: глобальность через namespace, простота контроля.

Сложность: **низкая** (реализуется DDL и приложениям несложно).

Минусы: увеличивает ширину индекса, сложнее внешние FK; некоторые ORM/инструменты плохо работают с составными PK.

5. **UUID как business_id, BIGSERIAL как PK**

Как работает: хранить `id BIGSERIAL PRIMARY KEY`, а `uuid UUID UNIQUE NOT NULL` для внешних интеграций/публичных ссылок (генерировать v4/v7).

Воспроизводит: уникальность для внешних потребностей, компактную индексируемую PK внутри БД.

Сложность: **низкая**.

Минусы: дублирование поля, необходимость поддерживать согласованность; JOIN по uuid чуть дороже.

6. **Sequence range allocation (preallocated ranges per node)**

Как работает: выделять каждому узлу диапазон чисел из глобальной последовательности (например, nextval выдало блок 1000–1999), узел использует их локально.

Воспроизводит: уникальность без per-insert сетевых запросов, компактность.

Сложность: **средняя**.

Минусы: возможны пропуски при падении узла; требуется сервис выдачи диапазонов.

Практическая рекомендация (какую альтернативу для каких случаев)

Архитектура / случай использования	Рекомендуемая альтернатива	Причина
Высоко-нагруженный OLTP в одном DC	Snowflake или preallocated ranges	компактность, упорядоченность, низкий I/O индексов
Распределённые микросервисы, offline-клиенты	ULID / KSUID или UUID как business_id + BIGSERIAL	локальная генерация + сортируемость/удобство API
Public API + приватность	UUIDv4 как business_id + BIGSERIAL PK	скрытие объемов, читабельность внутри БД
Legacy интеграции / аналитика	Composite (service_id + serial) или namespaced sequences	компактность, совместимость с внешними системами

Добавь этот вывод в Лог взаимодействия как запись №2

Если хочешь, могу сразу собрать пример DDL и snippets (PostgreSQL) для каждой альтернативы: `CREATE SEQUENCE`, `nextval()`, пример Snowflake-алгоритма на Go/Python, и UUID/KSUID генерацию.

Промпт 3

Теперь нужно получить чисто техническую часть – DDL-скрипты и сценарий нагрузочного теста, который демонстрирует различие в массовой вставке между таблицей с UUID PK и таблицей с BIGINT (BIGSERIAL) PK.

Часть А – DDL

1. Сгенерируй **два** полных, готовых к исполнению PostgreSQL DDL-скрипта (включая необходимые расширения), с учётом best practices:

- A1: Таблица `event_log_uuid` с PK типа UUID.
 - Используй рекомендованный способ генерации UUID в PostgreSQL (укажи варианты: `uuid-ossp` vs `pgcrypto` vs `uuid_generate_v1mc/uuid_generate_v6`), сделай выбор и объясни почему.
 - Добавь несколько столбцов: id PK, event_time (timestamp), user_id UUID, payload JSONB, created_at default now().
 - Добавь индексы, рекомендации по fillfactor, и дополнительно кластеризацию/партиционирование.
- A2: Таблица `event_log_bigserial` с PK типа BIGINT (BIGSERIAL или `GENERATED BY DEFAULT AS IDENTITY`).
 - Структура столбцов должна быть максимально аналогична `event_log_uuid` (user_id можно оставить UUID или BIGINT – объясни выбор).
 - Добавь те же индексы и рекомендации по настройкам.

2. В каждом DDL укажи: типы индексов, ограничения, рекомендации по autovacuum, fillfactor, на какие колонки кластеризовать и почему.

Часть В – сценарий нагрузочного теста

1. Предложи **реалистичный** нагрузочный сценарий, который можно запустить на тестовой машине (укажи аппаратные предположения: RAM, CPU, SSD vs HDD), параметры БД (shared_buffers, wal_level, synchronous_commit и т.п.) и предполагаемый объём данных.

- Рекомендуемые тестовые размеры: 1М, 5М, 20М строк (укажи почему).
 - Параметры конкурентных клиентов: 1, 8, 32.
 - Batch size при вставке: 1, 100, 1000.
2. Дай **пошаговый** набор команд/скриптов для выполнения теста:
- Скрипты/команды генерации данных (варианты: pgbench custom script, Python генератор + COPY, shell + openssl/uuidgen для uuid). Приведи пример Python-скрипта (psycopg2 или asyncpg) для генерации и массовой вставки в батчах с параметрами.
 - Альтернативный вариант: использование `pgbench` с custom script – сделай пример такого .sql-скрипта.

- Команды запуска теста (например: `pgbench -c 32 -j 8 -T 300 -f insert_uuid.sql` или инструкция как запустить Python-скрипт).

3. Какие метрики собирать и как измерять:

- Время полного INSERT (wall time).
- TPS (transactions per second) и rows/sec.
- Размер индекса и таблицы на диске (до/после).
- WAL generation (MB/s).
- VACUUM/ANALYZE потребности.
- Latency P50/P95/P99.
- CPU, IOwait.
- Снимки `pg_stat_all_tables` и `pg_stat_user_indexes`.

4. Ожидаемые результаты и объяснение причин:

- Опиши какие различия в производительности и размере индексов мы ожидаем и почему (включая факторы: размер ключа, локальность вставок, индексный b-tree bloat, random write vs append, сортируемые/ordered ids, cache efficiency).

- Укажи варианты UUID (v4 random vs v1/v6/ULID) и как они изменят картину; предложи эксперименты, которые захватят этот фактор.

5. Наконец: добавь инструкции по репликации результатов в CI (короткая заметка – какие артефакты со

хранять: raw logs, pg_stat snapshots, du output, pgbench reports).

6. Объём: DDL + сценарий вместе примерно 600-1200 слов + код-образцы.

7. В конце: "Добавь этот вывод в Лог взаимодействия как запись №3".

Требование: все SQL/скрипты должны быть в кодовых блоках, готовые к копированию/выполнению; пояснения – отдельными абзацами.

Ответ 3

Часть А — DDL

Ниже — два готовых к исполнению PostgreSQL DDL-скрипта и пояснения. Я предпочитаю использовать функции `pgcrypto`/`gen_random_uuid()` для повседневной генерации UUIDv4 (быстрее, доступно в стандартных пакетах, криптографически случайно). Для сценариев, где нужна time-ordered ID, рекомендую ULID/KSUID или `uuid-ossp`/`uuid_generate_v1mc()` / внешние либы, либо `uuid v6/v7` реализации (черновики) — см. комментарии в скриптах.

A1 — `event_log_uuid` (UUID PK, `pgcrypto`)

```
--sql
-- A1: event_log_uuid
CREATE EXTENSION IF NOT EXISTS pgcrypto; -- provides gen_random_uuid()

-- Schema
CREATE TABLE IF NOT EXISTS event_log_uuid (
    id      uuid PRIMARY KEY DEFAULT gen_random_uuid(),
    event_time timestampz NOT NULL,
    user_id  uuid NOT NULL,
    payload   jsonb,
    created_at timestampz NOT NULL DEFAULT now()
);

-- Indexes
-- PK on id (btree on uuid). For time-range queries, index event_time.
CREATE INDEX IF NOT EXISTS idx_event_log_uuid_event_time ON event_log_uuid (event_time);
CREATE INDEX IF NOT EXISTS idx_event_log_uuid_user_time ON event_log_uuid (user_id, event_time DESC);

-- Recommendations:
-- fillfactor lower for random UUIDs to reduce page splits (e.g. 70)
ALTER TABLE event_log_uuid SET (fillfactor = 70);

-- Partitioning suggestion (time-range) for large volumes:
-- CREATE TABLE event_log_uuid_y2025 PARTITION OF event_log_uuid FOR VALUES FROM ('2025-01-01') TO ('2026-01-01');

-- Autovacuum recommendations (example settings per-table via ALTER TABLE):
ALTER TABLE event_log_uuid SET (autovacuum_vacuum_scale_factor = 0.02, autovacuum_vacuum_threshold
= 50);
--
```

Пояснения A1

* Выбор `pgcrypto`/`gen_random_uuid()` (UUIDv4) — простая, быстрая генерация и широко доступна.

* Для high-insert workloads можно заменить на time-ordered IDs (ULID/KSUID или uuid v7) чтобы уменьшить индекс-фрагментацию.

* Fillfactor=70 уменьшает число page-splits при случайной генерации UUID; автовакуум настроен чаще, чтобы ограничивать bloat.

```

### A2 — `event_log_bigserial` (BIGINT PK)

```sql
-- A2: event_log_bigserial
CREATE TABLE IF NOT EXISTS event_log_bigserial (
 id bigint GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
 event_time timestampz NOT NULL,
 user_id uuid NOT NULL, -- выбираем UUID для user_id (унификация с UUID таблицей)
 payload jsonb,
 created_at timestampz NOT NULL DEFAULT now()
);

-- Indexes
CREATE INDEX IF NOT EXISTS idx_event_log_bigserial_event_time ON event_log_bigserial (event_time);
CREATE INDEX IF NOT EXISTS idx_event_log_bigserial_user_time ON event_log_bigserial (user_id, event_time DESC);

-- fillfactor can be higher for sequential inserts
ALTER TABLE event_log_bigserial SET (fillfactor = 90);

-- Autovacuum recommendations
ALTER TABLE event_log_bigserial SET (autovacuum_vacuum_scale_factor = 0.05, autovacuum_vacuum_thresh
old = 100);
```

```

Пояснения A2

- * `id` — `IDENTITY` (SQL-стандарт), предпочтительнее `BIGSERIAL` в новых схемах.
- * `user_id` оставлен `uuid` чтобы сравнение таблиц было честным (одинаковый size для user_id колонок).
- * Fillfactor высокая: последовательные вставки минимизируют page splits, лучше packing.

Часть В — сценарий нагружочного теста

Аппаратная и БД-конфигурация (предложение)

- * Машина: 8 CPU cores, 32 GB RAM, NVMe SSD (лучше) — если HDD, ожидать худших IO результатов.
- * PostgreSQL (пример, файл postgresql.conf):

```

```

shared_buffers = 8GB
work_mem = 32MB
maintenance_work_mem = 1GB
wal_level = replica
synchronous_commit = off -- для теста вставок можно отключить (показать best-case)
wal_compression = on
checkpoint_timeout = 10min
max_wal_size = 2GB
```

```

- * Предполагаемый объём данных: 1M, 5M, 20M строк — дают градуированные масштабы (локальная cache, рост индексов, видимая фрагментация).

Параметры тестов

- * concurrent clients: 1, 8, 32
- * batch sizes: 1, 100, 1000
- * этапы: для каждой комбинации (clients × batch) запускать тест на X минут или до вставки N строк.
- * Причина размеров: 1M — влезает в кэш частично; 5M — заметный рост индексов; 20M — реальные IO

эффекты.

Скрипты / команды

Вариант А — Python (psycopg2 + COPY) — быстрый и репрезентативный

```
```python
save as load_uuid_copy.py
import uuid, json, time, sys
import psycopg2
from io import StringIO
from datetime import datetime, timezone
conn = psycopg2.connect("host=127.0.0.1 dbname=test user=test password=test")
cur = conn.cursor()

def gen_batch_csv(batch_size, start_ts):
 buf = StringIO()
 for i in range(batch_size):
 id = str(uuid.uuid4())
 event_time = datetime.now(timezone.utc).isoformat()
 user_id = str(uuid.uuid4())
 payload = json.dumps({"i": i})
 created_at = datetime.now(timezone.utc).isoformat()
 buf.write(f"{id}\t{event_time}\t{user_id}\t{payload}\t{created_at}\n")
 buf.seek(0)
 return buf

def copy_batch_uuid(batch_size, iterations):
 for _ in range(iterations):
 buf = gen_batch_csv(batch_size, None)
 cur.copy_from(buf, 'event_log_uuid', columns=('id','event_time','user_id','payload','created_at'), sep='\t')
 conn.commit()

if __name__ == "__main__":
 batch = int(sys.argv[1]) # e.g., 1000
 iterations = int(sys.argv[2]) # total_rows = batch * iterations
 t0 = time.time()
 copy_batch_uuid(batch, iterations)
 print("Elapsed:", time.time()-t0)
 cur.close(); conn.close()
```

```

Запуск: `python3 load_uuid_copy.py 1000 1000` (вставит 1_000_000 строк).

Для `event_log_bigserial` — тот же CSV, но без колонки `id`, и `copy_from` с колонками `event_time,user_id,payload,created_at` — id генерирует БД.

Вариант В — pgbench custom (insert script)

`insert_uuid.sql`:

```
```sql
\set id random_uuid()
\set user_id random_uuid()
INSERT INTO event_log_uuid (id, event_time, user_id, payload, created_at)
VALUES (:id, now(), :user_id, '{"x": :rseed }', now());
```

```

Запуск:

```
```bash
pgbench -c 32 -j 8 -T 300 -f insert_uuid.sql -U test -d testdb
````
```

Аналогично для `event_log_bigserial`:

```
```sql
INSERT INTO event_log_bigserial (event_time, user_id, payload, created_at)
VALUES (now(), random_uuid(), '{"x": :rseed }', now());
````
```

Примечание: `pgbench` имеет `random_uuid()` макро только в новых версиях; можно использовать `\\set id :unixepoch`+client-side generator — альтернативно генерировать CSV и `COPY`.

Метрики и как собирать

- * Wall time полного INSERT: `time` обёртка или лог Python.
- * TPS / rows/sec: `pgbench` вывод; или rows_inserted / elapsed.
- * Размер таблицы/индекса:

```
```sql
SELECT pg_relation_size('event_log_uuid') AS heap, pg_total_relation_size('event_log_uuid') AS total;
SELECT pg_relation_size(indexrelid) FROM pg_index JOIN pg_class ON pg_index.indexrelid=pg_class.oid WHERE indexrelid = 'event_log_uuid'::regclass;
````
```

или shell: `du -h \$(pg_lsclusters ...)/base/...`

- * WAL: `select pg_current_wal_lsn();` diff before/after; or monitor `pg_wal` size.
- * VACUUM/ANALYZE: `vacuumdb --analyze` and `pg_stat_all_tables`.
- * Latency P50/P95/P99: collect via `pgbench -r` or external metrics (Prometheus + pgbouncer) or in Python measure per-batch.
- * CPU/IOWait: `top`, `iostat -x`, `vmstat 1`.
- * Snapshots:

```
```sql
SELECT * FROM pg_stat_all_tables WHERE relname LIKE 'event_log%';
SELECT * FROM pg_stat_user_indexes WHERE relname LIKE 'event_log%';
````
```

Ожидаемые результаты и объяснение причин

- * **Throughput**: `event_log_bigserial` обычно даст выше throughput и меньше I/O при вставках по сравнению с random-UUID, особенно при высокой concurrency и небольших batch (1/100). Причина — последовательные BIGINT вставкиappenд-ориентированы, минимальные page-splits, кэширование лидирующих страниц индекса.
- * **Индекс size**: индекс на UUID PK будет ~2× больше (16 byte keys vs 8 byte) плюс фрагментация → больше страниц → хуже cache hit.
- * **WAL**: random writes/индекс-расширения генерируют больше WAL, больше fsync/IO.
- * **Fillfactor/Autovacuum**: уменьшает эффект, но не ликвидирует.
- * **Варианты UUID**: UUIDv4 (random) — худший по вставкам; UUIDv1/v6/v7 или ULID/KSUID (time-order) уменьшают фрагментацию и поведение будет ближе к BIGSERIAL. Эксперимент: сравнить gen_random_uuid(), uuid_generate_v1mc(), и ULID (генерация на клиенте) при одинаковых нагрузках.

Репликация результатов в CI — что сохранять

* raw logs (`pgbench`/Python stdout), метрики CPU/IO (iostat, vmstat), `psql` outputs (`pg_relation_size`, `pg_stat_*`).

* `du -sh` результатов каталогов данных/индексов.

* `pg_wal` size / WAL delta.

* Команда `EXPLAIN ANALYZE` для representative queries.

Сохранять эти артефакты (архив) в CI job artifacts для последующего анализа.

Добавь этот вывод в Лог взаимодействия как запись №3

Если нужно — подготовлю ready-to-run CI job (GitLab CI / GitHub Actions) с контейнером PostgreSQL, генерацией данных и сбором артефактов.