PGMeetup.DBA

Типичные ошибки при работе с PostgreSQL

Фролков Иван

PostgresPro

PGMeetup.DBA

ТРАДИЦИОННЫЕ ТИПЫ

PostgresPro

* Тип NUMERIC/DECIMAL имеет переменную длину. Если у вас колонка типа DECIMAL, туда теоретически можно запихать очень большое значение. Всегда указывайте точность
* В Oracle есть тип number. Это отличный тип; при переходе с Oracle часто не глядя заменяют его NUMERIC. Все работает, но размер NUMERIC минимум 8 байт и относительно сложная конструкция внутри, что ведет к снижению производительности. Правило: NUMERIC - это для денег и только для денег
* Тип TEXT имеет длину до гигабайта (об этом чуть позже). Если колонка с email имеет тип TEXT, то туда без можно положить стомегабайтный email, что как-то странно. Длину надо ограничивать
* Есть тип UUID, и надо пользоваться им, а не текстовым представлением. Это и быстрее, и имеет меньший размер (16 байт, а не 33)
* COLLATION
  * Мы в России, сейчас везде UTF8, и Postgres не исключение. Все бы хорошо, но с UTF8 из коробки LIKE работает плохо. Что делать? Поставить расширение pg_tgrm; для всяких служебных идентификаторов можно поставить COLLATE "C", тогда сравнение будет побайтовым. На больших объемах можно получить заметный прирост производительности
* Выравнивание
  * Данные в строке выровнены на естественные позиции (1,2,4,8 байт). Размер строки: заголовок + выровненные колонки
  * Кстати, заголовок - теоретически 23-25 байт, практически - 32. Он тоже выравнивается

PGMeetup.DBA

НЕТРАДЦИОННЫЕ ТИПЫ

PostgresPro

* JSON/JSONB - не увлекайтесь JSON/JSONB, обычно это плохо заканчивается. Почему?
  * JSON толстый
    * Там хранятся и имена атрибутов. В итоге размер БД зависит от наименований атрибутов. Это глупо
    * А для фильтрации потребуется вычитать весь JSON. Это может быть весьма болезненной операцией, когда потребуется лезть в TOAST
  * JSON непрозрачный: даже для функциональных индексов статистики нет. Могут быть вопросы "а почему мой индекс не используется?"
  * При Bitmap Index Scan при больших объемах данных вместо ссылки на строку-кандидат может использоваться ссылка на страницу, в которой находится строка-кандидат (все равно в итоге надо читать страницу). Если в условии используется колонка JSON, и значения достаточно большие, то во время выполнения сервер поднимет из TOAST все строки со страницы, что обычно ведет к катастрофическому снижению производительности
  * JSON гибкий до бесформенности: в колонке с JSON может оказаться вообще что угодно. JSON SCHEMA, увы, не реализована.
  * Вывод: JSON надо использовать именно как JSON - для хранения ответов удаленных систем, как тело сообщения в системе передачи сообщений и т.п. Как основной способ хранения JSON не подходит (и не только в Postgres, подозреваю)

PGMeetup.DBA

BLOB

PostgresPro

* Два варианта - Large Object (старый и побыл obsolete, сейчас partially obsolete) и bytea (или text)
* Large object
  * отдельная таблица (одна, непартиционированная!)
  * отдельный объект БД (с правами и т.д.)
  * Есть улитита vacuumlo, удаляющая потерянные объекты
    * Она просматривает колонки с типом oid, и если на некоторый объект никакая такая колонка не ссылается - удаляет его
    * jBPM хранит ссылки на large object в колонках другого типа, будьте внимательны!
  * Зато размер - до 4T (таких может быть 8 штук на всю таблицу - ограничение 32T на таблицу)
  * Можно читать из середины
* bytea
  * 1Г (теоретически; практически может оказаться так, что его можно записать, но потом не прочитать)
  * Сжимается
  * Может храниться вне строки, может - внутри
  * Нельзя читать из середины, только целиком
* Выводы
  * Postgres плохо подходит для хранения больших двоичных данных
  * При желании все-таки можно

PGMeetup.DBA

НЕТРАДИЦИОННЫЕ ВОЗМОЖНОСТИ

PostgresPro

* JIT
  * JIT это хорошо, только вот на его компиляцию тоже тратится время, причем достаточно заметное; т.е. только отпрепаренные запросы. С отключенным JIT в большинстве случаев запрос будет работать, скорее всего, быстрее
* Параллельное выполнение.
  * Это тоже хорошо, но ведет в лучшем случае к кратному повышению производительности запросов (что хорошо, но мало) и увеличению конкуренции за некоторые внутренние структуры (что хуже). Для OLTP подходит плохо

PGMeetup.DBA

ТРАНЗАКЦИИ

PostgresPro

* Postgres это умеет и умеет хорошо. Надо этим пользоваться
  * Транзакционный DDL! Обычно при переходе с других СУБД об этом никто не подозревает, но это так.
    * В случае изменения функций могут быть чудеса, но для типовой ситуации "атомарно изменить набор функций" это отлично работает
    * С таблицами то же самое
    * VIEW зависят от VIEW, если есть цепочка VIEW->VIEW->VIEW, то базовый VIEW так просто поменять не получится. Пользутесь либо скриптами пересоздания (можно найти в сети), либо функциями
* Postgres не любит длинные транзакции
  * Что такое длинная транзакция? Вопрос философский
  * Почему не любит? Вакуум не может удалить мусор, появившийся с начала транзакции
  * Почему они возникают?
    * Логика работы приложения. Иногда это надо
    * Чаще всего - потерянные соединения
      * Чините приложения!
      * Если не чинится - idle_in_transaction_session_timeout
  * Что делать?
      * Не устраивать длинные транзакции, разбивать задачу на несколько коротких
      * Убивать слишком длинные транзакции

PGMeetup.DBA

PostgresPro

2PC

* По умолчанию выключен
* Редко пользуются, а зря:
  * Перенос данных между системами
  * В ряде случаев можно иметь две (три, четыре...) копии базы без всяких репликаций
  * **Корректная многопоточная транзакционная обработка**
* postgres_fdw (и dblink) без 2PC – обратите внимание!

PostgresPro

SAVEPOINT ИЛИ ТОЧКИ СОХРАНЕНИЯ ИЛИ ПОДТРАНЗАКЦИИ

* Отличная возможность
* Есть неприятные ограничения
* Не более 64 на транзакцию
  * Можно и больше, но не нужно
  * Почему? Потому что вытесняются из кеша сессии
  * Типовой случай: поставили savepoint, попытались добавить строку, получили ошибку, откатились, обновили
    * Ну, если занудствовать - это некорректно. Строку могут и удалить между ошибкой и обновлением
    * Проблемы с производительностью
    * Если один-два раза - скорее всего, ничего страшного
    * insert .. on conflict update / merge (с 15 версии)

PGMeetup.DBA

PostgresPro

ВРЕМЕННЫЕ ТАБЛИЦЫ

* Отличная вещь, но...
  * Это, строго говоря, обычная таблица – занимает место в системном каталоге; надо чистить мусор
  * Если раз в час или в пять минут - ничего страшного
  * А вот несколько сотен раз в секунду - готовьтесь к проблемам
  * Кстати, не работает на реплике
  * Что делать?
    * Массивы
    * pg_variables
    * Внешнее хранение

PGMeetup.DBA

PostgresPro

СЕКЦИОНИРОВАНИЕ, ОНО ЖЕ ПАРТИЦИОНИРОВАНИЕ

* **Всегда** откладывают на потом
  * Партиционирование терабайтной таблицы онлайн - довольно сложное занятие
  * Делайте сразу!
  * Максимальный размер таблицы - 32T и достигается внезапно. Выкрутиться можно, но лучше обойтись без этого
  * Особый случай - OLTP, популярная таблица, много физических ядер
    * Страшная конкуренция за корень индекса/первую страницу
    * Партиционирование может помочь
* Глобальных индексов нет
  * Только вспомогательные таблицы
  * IOT-таблиц тоже нет, кстати

PGMeetup.DBA

ИНДЕКСЫ

PostgresPro

* Традиционные btree
  * См. выше про collation
  * Условные индексы (create index on t(id) where is_sold). Иногда - просто спасение
  * Индексы по функциям. Функции - только immutable (надо только пообещать серверу, а так делай что хочешь. Если захочешь. Но лучше не надо). Иногда тоже спасают
  * Index skip scan нет.
  * Index scan vs bitmap index scan
* Нетрадиционные
  * GIN - индексы по массивам. Полнотекстовой поиск, триграммы. Нужны нечасто, но нужны
  * GIST/SpGIST - в основном география или диапазоны

PGMeetup.DBA

VACUUM

PostgresPro

* Сборка мусора - удаленные строки. Ну не только удаленные, но особой разницы нет
* Замусориваются не только таблицы, но и индексы. Чисто не там, где убирают, а там, где не сорят
* **Автовакуум ОТКЛЮЧАТЬ НЕЛЬЗЯ НИ В КОЕМ СЛУЧАЕ!**

PGMeetup.DBA

СПАСИБО

postgrespro.ru