https://habr.com/ru/post/210920/

Хранимые функции. За и против

Oracle *Программирование *

Использование хранимых функций СУБД для реализации бизнес-логики или её части, всегда было камнем преткновения. С одной стороны баррикад DBA и программисты БД, с другой — разработчики backend.
Рискну навлечь на себя гнев из обоих лагерей, но всё же просуммирую плюсы и минусы и изложу свои соображения о том, когда стоит писать код в хранимых функциях, а когда следует выносить наружу.

Начнём с аргументов против:

Размазывание бизнес-логики

Это, на самом деле не проблема СУБД и ХФ, как инструмента — это проблема их неверного использования. У программиста бд может возникнуть желание описать всю логику реализуемого действия в хранимой функции — действительно, ведь все данные вот они, под рукой. Если программист поддастся на искушение, а его руководитель не возразит, в будущем могут возникнуть проблемы с узостью интерфейса со внешней системой (например, с сервером приложений) — придётся добавлять новые параметры, усложнять логику и т.п. Это даже может привести к тому, что появятся «дублирующие» ХФ со слегка иным функционалом.

Скудность языка СУБД

Есть такое дело. Традиционные языки для написания ХФ pl/sql, t-sql, pl/pgsql довольно примитивны по сравнению с современными языками общего назначения. Стоит заметить, что есть возможность писать ХФ и на более продвинутых языках, например Java в Oracle или Python в postgresql.

Непереносимость хранимых функций

Имеется в виду несовместимость диалектов процедурных языков разных СУБД. Многоплатформенность как раз на уровне — благодаря поддержке разных ОС и архитектур в самих СУБД и независимости встроенных языков от внешней платформы. Здесь опять решение зависит от специфики проекта. Если проект тиражируемый, причём вы не контролируете платформу (классический пример — CMS), то переносимость вам необходима и использование ХФ — только добавит головной боли. Если же проект уникальный, либо внедрения будут происходить унифицировано (например в разных филиалах одной компании), то про непереносимость между разными СУБД можно забыть.

Отсутствие необходимых навыков у команды и высокая «стоимость» соответствующих специалистов

Это, на мой взгляд, самый серьёзный аргумент против использования ХФ. Тут всё зависит от масштабов проекта. Грубо говоря, использование хранимого кода на стороне СУБД оправдано в средних-крупных enterprise проектах. Если проект помельче — овчинка выделки не стоит. Если проект огромный сверхнагруженный, то архитектура с ХФ и РСУБД упрётся в проблемы масштабирования — тут необходимо использование специфического хранилища и подхода к обработке данных.

Теперь плюсы:

Скорость

При обработке даже небольших объёмов данных во внешнем приложении мы тратим дополнительное время на передачу по сети и преобразование данных в нужный нам формат. К тому же в СУБД уже встроены, отлажены и протестированы близкие к оптимальным алгоритмы обработки данных, вашим программистам незачем практиковаться в изобретении велосипедов.

Сокрытие структуры данных

С ростом и эволюцией программной системы схема данных может и должна меняться. Хорошо спроектированный программный интерфейс на ХФ позволит менять схему данных не изменяя код внешних приложений (которых может быть несколько). Отсюда органично вытекает и разделение ролей разработчиков, которые работают с БД и знают её структуру, и разработчиков внешних приложений, которые должны знать лишь предоставляемый API. При использовании динамического SQL на стороне приложения, для подобного разделения вводятся дополнительные слои программных абстракций БД, различные ORM.

Гибкое управление правами доступа

Хорошей практикой является ограничение пользователя, под которым «ходит» в базу клиентское приложение в правах таким образом, что он не имеет прав на чтение и изменение никаких объектов. Лишь выполняет разрешённые ему функции. Таким образом можно жёстко контролировать какие действия доступны клиенту, уменьшается вероятность нарушения целостности данных из-за ошибки клиентского приложения.

Меньшая вероятность SQL injection

При использовании динамического SQL со стороны клиентской программы, клиентская программа передаёт СУБД SQL команды в виде строк, предварительно формируемых в коде. При формировании этих строк программисту нужно быть предельно внимательным, чтобы не допустить возможности непредусмотренной модификации SQL команды. При использовании ХФ

SQL код на стороне приложения обычно статический, и выглядит, как простой вызов ХФ, параметры которой передаются не строками, а через placeholders (:variable) через механизм binding. Конечно это не исключает возможность SQL injection полностью (ведь можно умудриться в ХФ конкатенировать строку, переданную параметром с текстом динамически выполняемого SQL запроса), но значительно уменьшает её вероятность.

Повторное использование SQL

Реализуя логику работы с данными в хранимом слое, мы получаем привычную нам иерархическую модель повторного использования SQL кода.
При использовании динамического SQL повторное использование запросов затруднено. Например пусть есть система А на базе ХФ и система Б на базе динамического SQL. В обеих системах есть функция получения цены товара get_price. В случае А — это хранимая функция или отображение (view), в случае Б, допустим, процедура на java, через JDBC выполняющая SQL запрос. Есть задача — получить общую стоимость товара на складе. В случае А мы джоиним get_price прямо в запрос, получающий список товаров на складе (в случае, если get_price — view или ХФ на SQL, как например в PostgreSQL, то оптимизатор разворачивает запрос inline — тем самым получается один запрос, который быстро находит сумму).
В случае В есть два варианта — либо пробежать по курсору с выборкой товаров на складе и n раз вызвать get_price (а это значит что вся выборка должна передаться по сети на клиент) либо забыть про повторное использование и написать подзапрос, дублирующий тот, что был уже написан в get_price. Оба варианта — плохие.

Простая отладка SQL

Упрощается отладка (по сравнению с разнородной процедурой внешний код+sql)
В системах с динамическим SQL (любые ORM) даже простая задача поиска проблемного куска SQL может оказаться сложной.
Семантическая и синтаксическая проверка SQL на этапе компиляции.
Возможность профилирования функций и поиска узких мест.
Возможность трассировки уже запущеной и работающей системы.
Автоматический контроль зависимостей — при изменении определения объекта инвалидируются зависимые сущности.

Когда писать бизнес-логику в БД?

Если важна скорость обработки данных

Обработка данных прямо на месте их хранения зачастую даёт значительный прирост скорости обработки. Становятся возможными такие оптимизации, как, например, агрегации на уровне хранилища данных — данные с массива даже не передаются на сервер СУБД, не говоря о клиенте.

Когда важна целостность и непротиворечивость данных

В хранимых функциях с явным управлением транзакциями и блокировками проще обеспечить целостность данных и атомарность операций. Конечно всё это может быть реализовано и снаружи, но это отдельная и большая работа.

Данные имеют сложную, но устоявшуюся структуру

Плоские и слабо взаимосвязанные структуры часто не требуют всего богатства инструментов обработки, которые предлагают СУБД. Для них можно использовать сверхбыстрые key-value хранилища и кеширование в памяти.
Сложно организованные сильно связанные иерархические и сетевые структуры — явный показатель, что ваши знания РСУБД пригодятся!

Когда выносить код наружу?

Работа с внешними данными

Если специфика системы такова, что данных, приходящих на обработку снаружи (с датчиков, из других систем) больше, чем данных, сохраняемых в БД, то многие плюсы БД, как платформы программирования теряются. Оказывается, проще обработать поступающие данные снаружи и сохранить результат в БД, чем сначала всё пихать в БД, а потом обрабатывать. Здесь соблюдается тот же принцип — обрабатывать данные как можно ближе к источнику, о котором мы говорили выше применительно обработке данных, уже хранящихся в БД.

Сложные алгоритмы

Сложные или высоко-оптимизированные алгоритмы-числодробилки лучше писать на более приспособленных для этого языках. Встроенные языки РСУБД очень мощны (в том смысле, что высокоуровневые, а не гибкие), но за счёт этого имеют высокий overhead.

Highload

В сверхвысоконагруженных системах обычные подходы к сериализации транзакций и синхронизации серверов кластера становятся узким местом. Для таких систем характерны уникальные решения под конкретные задачи, универсальные и мощные системы РСУБД часто оказываются слишком медлительными при нагрузках в сотни тысяч конкурентных транзакций в секунду.

Вывод такой, что чёткого алгоритма нет. Каждый раз решение остаётся за архитекторами и менеджером и от него зависит то, завязнет ли проект в проблемах с race conditions и неконсистентностью данных NoSQL, проблемах с производительностью и отладкой запросов ORM, или упрётся в проблемы масштабирования СУБД при использовании хранимых функций. Поэтому — принимайте верные решения :)