КЗСУБД (Кэтрин)

Фунциональность:

1. Сетевая/внедряемая (*).
2. Простой доступ из языка программирования, удобная работа с единичными объектами.
3. Атомарность/ACID.
4. Использование только передовых технологий. Многопоточность, возможности ОС.
5. Работа с множествами и парами ключ/сложный объект.
6. Объектное хранение и модификация схемы.
7. Автоматическая агрегация.
8. Свободная лицензия LGPL3 для использования в любых целях.
9. Понятная перспектива поддержки SQL или его аналогов, клиент-серверного варианта.

(*) Выделенное по тексту маркером не реализовано либо подлежит переработке.
Структуру файлов, краткое описание и release notes см. в k3dbms\readme.txt

Цели эксперимента:

1. Проверить скорость работы с NTFS/ TxF, как структурной ФС. (Возможности похожи на внутреннюю реализацию в БД)
2. Исследовать факторы, влияющие на производительность – блокировки файловые и межпроцессные, кэш, потоки, RAID.
3. Проработать способы реализации SQL-подобной работы со множествами.
4. Проработать детали локализации (юникод, сортировки).
5. Максимально использовать (параллельную) STL, Boost

Определения базисных понятий

Объект, сущность с атрибутами по стандарту JSON:

- Number (integer, real, or floating point)
- String (double-quoted Unicode with backslash escaping)
- Boolean (true and false)
- Array (an ordered sequence of values, comma-separated and enclosed in square brackets)
- Object (collection of key:value pairs, comma-separated and enclosed in curly braces)
- Null
- BLOB (эквивалентен Array byte, но хранится и обрабатывается отдельно от остальной части объекта)
- Link *Ссылка* на объект

Ссылка на объект – пара (тип объекта, *UID*)

UID – уникальный ключ объекта, позволяющий из *множества* объектов быстро получить требуемый. После удаления объекта его UID не присваивается новому объекту.

Множество – множество объектов одного типа. Для реализации индексного доступа и операций над множествами генерируются *зависимые* множества объектов {Indexkey, UID} и декартовы произведения{Link(obj1.UID), Link(obj2.UID),…, Link(objN.UID)}

Зависимое множество – множество, подмножество объектов которого взято из другого множества. При обращении зависимое множество перестраивается.

Агрегированное множество – зависимое множество, генерируемое на основе объектов исходного множества. В основном для целей аналитики.

Операции с объектами

Загрузка объекта, Вставка или Изменение, Удаление. Выгрузка в XML-JSON. При сохранении выполняется обновление зависимых множеств.

Изменение структуры объекта. Объект имеет поле «версия схемы» и при загрузке должен сам понимать как десериализоваться в нужной версии.

Поиск объектов

Пример: Множество школьников из школы №9, класса 3в, не носящих очки.
Возможно несколько вариантов простого поиска:

1. Загрузка единичного экземпляра по уникальному ключу;
2. Возвращается список UID подходящих объектов по неуникальному ключу;
3. Для каждого объекта по диапазону первичного ключа вызывается функция пользователя (callback or functor);
4. Создается зависимое множество для дальнейших операций.;

Операции с множествами

На базе каждого множества может создаваться зависимое множество, как правило – ссылок, но может содержать и дополнительные данные, например агрегаты.
Для зависимых множеств применимы операции с множествами см раздел внизу.
Умножение множеств = JOIN, отношение подмножество (Master-Detail) = умножение объекта на подмножество.

Если зависимое множество при создании имеет имя, то оно сохраняется, если нет – храним в памяти.
По фиксации транзакции производное множество сохраняется на диск, нет так нет.

Реализация

Хранение данных

Данные храним поверх NTFS. Transactional NTFS (TxF) дает A&D + Isolation уровня RC.
Плюсы такого подхода – экстенты, сжатие, сетевые кластера, обеспечение безопасности и прав доступа, гарантированная надежность, транзакции (в т.ч.распределенные), масштабируемость - многопроцессный доступ, БЛОБы, файловые и сетевые блокировки.

Множество = каталог, в нем блоки-файлы, разложенные по хеш-подкаталогам, максимум допустимо 2^32-1 файлов на том. Если хеш считать по ключевому полю – будет мастер индекс с упорядоченным обходом.
Блоки менее 600-700 байт будут храниться исключительно в MFT.
Если объекты мелкие(как правило), хеш-функция имени файла+каталога должна их объединять в один блок. Объекты могут быть переменного размера, потому в блоках нужно создавать резерв под «утолщение» объекта.

Опыт показал, что размер блока в 16Кб в несколько раз выгоднее по скорости, чем при 620байт. Поскольку типовой размер в RAID 64K – это типовой размер блока. Тем не менее, распределение объектам по блокам зависит от хэш-функции, и потому точный размер блока не соблюдается.

Структура файлового блока.
В начале блока CRC32, затем сортированное по ключу оглавление {кол-во элементов, {(хэш-ключ, смещение)}}, затем объекты.

Хэш-ключ (имя файла)
Поскольку NTFS на одном томе может хранить максимум 2^32 файлов, то хэш достаточно иметь 32-бит, но так как для уникальной идентификации объектов зачастую применяем GUID, применяем 128-битный ключ.
Для упорядочивания по ключу определяем хэш так: числа как есть, строки из Юникода приводим к однобайтовому представлению, (рус/лат минус ‘а’) и берем начальные байты (т.е.сортировка ключа из строк учитывает только 16 первых байт). Если размер записи мал, типовая хэш-функция делит ключ на (620(размер блока)/размер объекта).
Имя файла – строковое представления хэш-ключа в 16-тиричном формате.

Хэш-ключ (имя подкаталогов).
NTFS использует btree+ для поиска файлов. Чтобы сильно ей не мешать, имя каталога только 2 старших байта 0x16 (0000,..FFFF). В каждом каталоге будет не более 65’535 файлов.
Если заранее известно, что объектов будет немного, лучше все хранить в одном каталоге. FullScan по множеству – это обход дерева файлов. Поскольку NTFS сортирована, то при RangeScan читать все блоки необязательно, и даже возможно пропускать сканирование ненужных поддеревьев-каталогов.

Каталоги БЛОБов отдельные. Имя БЛОБА = имя хэш точка номер поля.Пока БЛОБы отложим (можем хранить и так прилично).

Метаданные

Метаданные – описание структуры объектов и множеств(файлов). XML или JSON с перечнем таблиц и параметров. Системные таблицы – таблицы, индексы, целостность и агрегаты.
В метаданные попадут имена множеств, а значит и зависимых множеств и индексов. Поскольку имена полей не используются во множествах, то на них данных нет.
Для реализации достаточно JSON или XML файла.
У объектов, а не только у экземпляров, должна быть описана структура с идентификатором в ОМД. А также должны быть описаны все связи и тоже с ИД.
TODO:Как вносить изменения в этот файл, чтобы видеть между транзакциями? Добавили зависимое именованное множество – переписали?!

Кэш и дисковые операции

Для каждого множества внутри каждой транзакции создается собственный кэш, синхронизация которых происходит по алгоритму когерентности write-once.

Изоляция. Стандартно без дополнительных блокировок TxF обеспечивает только RC-изоляцию, а при работе с XP и 2000 –только RU. Если TxF блочить на запись набор при чтении и запоминать список строк, то достигается RR или SR. Ограничение “TxF Multiple

transactions cannot concurrently modify the same file" – приводит к блокировкам на уровне блока данных. Подробнее по ссылке http://msdn.microsoft.com/en-us/library/dd979526(v=VS.85).aspx , абзац «Transactional Locking».

Для возможности распределения по шпинделям и оптимизации, каждое множество принадлежит объекту дисковой группы с индивидуальными настройками.
В дискгруппе используется параллелизм операций, например RAID1 = 2xЧтение,1xЗапись. Задавать его можно независимо для каждого множества, поскольку они могут быть на разных шпинделях.

В дискгруппе реализована очередь чтения/записи - getReader возвращает свободный reader (с меньшей очередью).
readObj проверяет в буфере записи!, потом своего кэша, потом с диска

Поток Writer. Очередь с ограниченной длиной? И потоки записи.
writeObj изменяет объект в очереди, а если нет, ставит в очередь.
TODO: Конфликт с разными транзакциями, размер записи!=размер блока. Транзакция сбрасывает грязные блоки.

Почему не общий буфер кэша/записи для разных транзакций? Невозможно, т.к.от файловой системы TxF для разных транзакций можем получить разные данные. Храня список изменений, можно сделать workaround, но тогда невозможна многопроцессная реализация доступа.

Алгоритм оптимального вытеснения кэша пока не реализован, хотя LRU ведется. FullScan множества опционально идет мимо нашего кэша, а остальные операции при заполнении вытесняют произвольные блоки.

Ведем статистику кэша, физических чтений, памяти итд.

Возможна реализация NEAR-Realtime, если создавать транзакции с конечным таймаутом, то неуспевшие будут откатываться автоматически (сейчас нереализовано и пенальти в производительности).

Структура объекта

Для уникальной идентификации объектов применяем GUID, т.е UID==GUID, хотя первичный ключ можем уникалить и обычным Thread-safe генератором.

В начале каждого объекта однотипный заголовок {размер объекта, UID, версия схемы, первичный ключ}. В пределах своего размера в памяти объект сам определяет сериализацию. Простейший способ, если объект типа POD (!string не является POD)– сырое копирование памяти.
Если объект содержит поля переменной длины, то рекомендуемый подход, например для типа Object (в реализации C++ map<>) {размер всего поля, количество эл-тов, {элемент, …}}, где элемент – пара тоже может быть переменной длины.
Для сериализации имеются готовые шаблоны.

Ссылка на объект = пара «имя множества»: «UID».

Ссылка на объект хранит только заголовок {ключ, UID}. При обычном обращении/поиска объекта, используется только стандартный заголовок, а объект десериализуется только по

необходимости загрузки. Исключение – Scan по индексу для проверки целостности индекса грузит весь объект.

Поиск объектов

Поиск по атрибуту Query = генерация подмножества, или FF/FN (NTFS sorted). Может использовать индексы, кроме поиска подстроки.
Задача искать по одному индексу, а сортировать по другому и сложных условий. Если у нас один индекс и несколько условий для скана – просто идем по индексу и проверяем остальные условия, если два и более индексных условий – строим общий для них битмап или список блоков и по этому списку пойдем (генерация списка - отключаемая для запроса возможность – тогда используем первый индекс в запросе). Сортировка возможна только по первому индексу или же в памяти – при возврате списка.

Как найти начало множества? FF
Конец множества и MAX – только FS
Как организован Fullscan таблицы? FF/FN+subcatalog.

Зависимые множества. Целостность

Сетевая реализация БД и сериализация требуют файловых блокировок. Для RR необходимо блокировать каждый файл-блок перед чтением, а снимать блок по закрытию транзакции. Для Serializable дополнительно в транзакции хранить список файлов, к которым обращались.

Циклическая проблема зависимых множеств. Ссылочной целостностью пока не занимаемся, кроме индексов.

Зависимое множество – это потомок множества с условиями ограничения подмножества и функтором построения зависимых объектов. Группировки и снежинки?

Реализация.Сначала отрабатываем условия ограничения – создаем все подмножества, потом умножаем = идем по тому подмножеству, которое упорядочивает и последовательно умножаем, получаем индекс (а,б). Если больше двух в операции, рекурсивно.

Индекс = подмножество, генерация индекса есть одна из операций. Индекс = мно-во = файл (ссылок на первичный ключ).Блокируется и модифицируется вместе с таблицей. Составные реализуются элементарно.

Частичное индексное покрытие. В зависимом множестве (например индексе) можно хранить выборочные поля, например для агрегатов.

Ленивый пересчет зависимых множеств. Пока нет чтения из него, храним список объектов, модифицирующих его, в памяти. Каждое 100е изменение приводит к пересчету всей пачки. Как с транзакциями?

Элементы OLAP. Достигается созданием зависимых множеств и индексов с частичным покрытием. Пример структуры базы в приложении ниже.

Проблемы и todo

Проблемы

1. Сложно предсказать заранее, равномерная ли заполняемость индекса и соответственно, размер файлового блока. Никак, только если ключ - генератор.
2. В блоке линейный поиск по UID и грузим весь в память - нельзя работать с очень большими блоками (многомегабайтными). Зато это аналог многоблочного чтения, если KEY == UID можно обходить.
3. Сложно отследить целостность, отслеживание связей (особенно без метаданных), M:M??
4. Непонятна идеология, что есть результат выполнения запроса. В нашем случае сложным запросом является процедура обработки данных, в которой используется навигация. Простой запрос – список объектов, удовлетворяющих условию.
5. Глобальная блокировка кэша при чтении – записи в кэш.

Приложения

(содержат выдержки из Википедии и sql.ru)

Маркетинг. Преимущества

Новая бесплатная БД – К3 (Кэтрин). ЗЫ. Про "новая и бесплатная" - это конечно шутка, которую пока разъяснять не буду.

1. Простой доступ из языка программирования, удобная работа с единичными объектами.
2. RENEW Отсутствие необходимости «привязывать» сущности прикладной области к реляционным таблицам, или создания маппингов ORM;
3. ACID;
4. Скорость работы выше, чем у ORM;
5. Прозрачная многопоточность внутри приложения;
6. Многоуровневое кэширование;
7. Эффективно хранит данные – на диске файлы данных могут занять меньше места, чем их реальный размер; (*)
8. RENEW.Работа с множествами связанных объектов. Поддержка объектов сложной структуры согласно JSON, со свойствами переменной длины – массивами и списками;
9. Компактный размер и простота реализации;
10. RENEW. Широкие возможности по оптимизации структуры хранения данных, индивидуальные настройки кэширования множеств и индексов.
11. RENEW. Автопересчет зависимых множеств, элементы OLAP.
12. Использует передовые технологии Windows Server 2008, Vista и Windows 7:
    12.1.Эффективное использование памяти;
    12.2.Масштабируемость по памяти, процессорам, дисковой емкости;
    12.3.Высококонкурентный многопроцессный доступ к файлам БД;
    12.4.Возможности NTFS V6.0 - экстенты, сжатие, безопасность, надежность, транзакции;
    12.5.Volume Shadow Copy безопасно для целостности копии БД;
13. Перспективы реализации (**) – кластерные системы, распределенные транзакции, сетевой-файловый конкурентный доступ, высокие уровни трансизоляции, БЛОБы, параллелизм дисковых операций;
14. Свободная лицензия LGPL3 для использования в любых целях.

(*) Законы сохранения никто не отменял, но есть трики.
(**) Чем сложнее, те более дальние.

ACID

Atomicity — Атомарность
Основная статья: Атомарность

Атомарность гарантирует, что никакая транзакция не будет зафиксирована в системе частично. Будут либо выполнены все её подоперации, либо не выполнено ни одной. Поскольку на практике невозможно одновременно и атомарно выполнить всю последовательность операций внутри транзакции, вводится понятие «отката» (rollback): если транзакцию не удаётся полностью завершить, результаты всех её до сих пор произведённых действий будут отменены и система вернётся в исходное состояние.
[править]
Consistency — Согласованность
Основная статья: Консистентность данных

Одно из самых сложных и неоднозначных свойств из четвёрки ACID. В соответствии с этим требованием, система находится в согласованном состоянии до начала транзакции и должна остаться в согласованном состоянии после завершения транзакции. Не нужно путать требование согласованности с требованиями целостности (integrity). Последние правила являются более узкими и, во многом, специфичны для реляционных СУБД: есть требования целостности типов (domain integrity), целостности ссылок (referential integrity), целостности сущностей (entity integrity), которые не могут быть нарушены физически в силу особенностей реализации системы.

Согласованность является более широким понятием. Например, в банковской системе может существовать требование равенства суммы, списываемой с одного счёта, сумме, зачисляемой на другой. Это бизнес-правило и оно не может быть гарантировано только проверками целостности, его должны соблюсти программисты при написании кода транзакций. Если какая-либо транзакция произведёт списание, но не произведёт зачисление, то система останется в некорректном состоянии и свойство согласованности будет нарушено.

Наконец, ещё одно замечание касается того, что в ходе выполнения транзакции согласованность не требуется. В нашем примере, списание и зачисление будут, скорее всего, двумя разными подоперациями и между их выполнением внутри транзакции будет видно несогласованное состояние системы. Однако не нужно забывать, что при выполнении требования изоляции, никаким другим транзакциям эта несогласованность не будет видна. А атомарность гарантирует, что транзакция либо будет полностью завершена либо будет полностью откачена, тем самым эта промежуточная несогласованность является скрытой.
[править]
Isolation — Изоляция
См. также: Уровни изолированности транзакций.

Во время выполнения транзакции другие процессы не должны видеть данные в промежуточном состоянии. Например, если транзакция изменяет сразу несколько полей в базе данных, то другой запрос, выполненный во время выполнения транзакции, не должен вернуть одни из этих полей с новыми значениями, а другие с исходными.
[править]
Durability — Долговечность

Независимо от проблем на нижних уровнях (к примеру, обесточивание системы или сбои в оборудовании) изменения, сделанные успешно завершённой транзакцией, должны остаться сохранёнными после возвращения системы в работу. Другими словами, если пользователь получил подтверждение от системы, что транзакция выполнена, он может быть уверен, что сделанные им изменения не будут отменены из-за какого-либо сбоя.

Операции над множествами

Бинарные операции

Ниже перечислены основные операции над множествами:
пересечение:

объединение:

Если множества A и B не пересекаются: , то их объединение обозначают также: .
разность:

симметрическая разность:

Декартово или прямое произведение:

Для лучшего понимания смысла этих операций используются диаграммы Эйлера — Венна, на которых представлены результаты операций над геометрическими фигурами как множествами точек.

Унарные операции
Абсолютное дополнение:

Операция дополнения подразумевает некоторый универсум (универсальное множество U, которое содержит A):

Относительным же дополнением называется A\B (см.выше):
Мощность множества:
| A |

Результатом является кардинальное число (для конечных множеств — натуральное).

Уровни изоляции

Стандарт SQL-92 определяет уровни изоляции, установка которых предотвращает определенные конфликтные ситуации. Введены следующие четыре уровня изоляции:
[править]
Serializable (упорядочиваемость)

Самый высокий уровень изолированности; транзакции полностью изолируются друг от друга. На этом уровне результаты параллельного выполнения транзакций для базы данных совпадают с последовательным выполнением тех же транзакций (по очереди в каком-либо порядке).
[править]
Repeatable read (повторяемость чтения)

Уровень, при котором чтение одной и той же строки или строк в транзакции дает одинаковый результат. (Пока транзакция не завершена, никакие другие транзакции не могут модифицировать эти данные.)
[править]
Read committed (чтение фиксированных данных)

Принятый по умолчанию уровень для SQL Server. Завершенное чтение, при котором отсутствует черновое, "грязное" чтение.(т.е. чтение одним пользователем данных, которые не были зафиксированны в БД командой COMMIT) Тем не менее в процессе работы одной транзакции другая может быть успешно завершена и сделанные ею изменения зафиксированы. В итоге первая транзакция будет работать с другим набором данных. Это проблема неповторяемого чтения.

В Oracle блокировки на чтение нет, вместо этого «читающая» транзакция получает ту версию данных, которая была актуальна в базе до начала «пишущей».
[править]
Read uncommitted (чтение незафиксированных данных)

Низший уровень изоляции, соответствующий уровню 0. Он гарантирует только физическую целостность данных: если несколько пользователей одновременно изменяют одну и ту же строку, то в окончательном варианте строка будет иметь значение, определенное пользователем, последним изменившим запись, а не смешанные значения столбцов отдельных пользователей (повреждение данных). По сути, для транзакции не устанавливается никакой блокировки, которая гарантировала бы целостность данных.
[править]
Поведение при различных уровнях изолированности

«+» — предотвращает, «–» — не предотвращает.

Уровень изоляции	Фантомная вставка	Неповторяющееся чтение	«Грязное» чтение	Потерянное обновление
SERIALIZABLE	+	+	+	+
REPEATABLE READ	–	+	+	+
READ COMMITED	–	–	+	+
READ UNCOMMITTED	–	–	–	+ (?)

12.02.2010– дата создания идеи К3

Тестовая задача «Про самолеты»

Alexey Rovdo
Member

Откуда: Москва
Сообщений: 913 Чтобы больше не поднимать эту тему, хотелось бы пояснить по поводу "подходимости" задачи для ОМД и РМД. Действительно существует большой пласт задач, которые больше подходят к РМД - и это именно те задачи, в которых работать приходится с табличными данными (бухгалтерия, финансовый учет и т.п.). Лично я не стану спорить с тем, что в реальной жизни таких задач встречается очень много (иначе и РМД не была бы так популярна). Но тенденции таковы, что требования к информационным системам постоянно усложняются, а вот при достижении определенного уровня сложности объектные методы становятся также вполне востребованными.
Однако рассматривать чересчур сложную задачу здесь не удастсяю Именно поэтому приходится подумать и предложить тему, которая с одной строны бы требовала генерации каких-то табличных отчетов. а с другой - предполагала бы и обработку неких сущностей, имеющих объектный характер (фифти-фифти).

Я предлагаю следующий (более абстрактный и более детально описанный пример):

Пусть есть города A, B, C .
Между ними по известному расписанию совершаются авиарейсы:
1) A -> B -> C -> A
2) C -> B -> A -> C

Для простоты будем считать, что рейсы длятся достаточно мало и отслеживать время отправления/прибытия нам не нужно - работаем только с датой.

Расписание имеет случайный характер, т.е. оно есть в виде таблицы:
дата - рейс - да/нет

Рейсы совершаются на разных видах самолетах, в которых есть разное количество мест 1-го и 2-го класса (R1(M1..Mn), R1(Mn+1...Mk), R2(M1...Mi), R2(Mi+1...Mj)).

Цена билета определяется по формуле:
S = f(X1,...Xn, Y, [Z1, Z2], r, h, d), где
X1...Xn - заданные заранее тарифные константы
Y - класс места
[Z1, Z2] - маршрут ([A,B], [A,C] ...)
r - рейс
h - количество одновременно покупаемых билетов
d - дата

Билет содержит:
цену S,
номер рейса r,
номер места m
маршрут [Z1, Z2]
имя пассажира Name .

Билет можно забронировать.
Информацию о проданных билетах нужно хранить.
Информацию об отмене бронирования хранить не нужно.
Возможные пересадки не учитываем.

Задачи

Дано: дата d, маршрут [Z1, Z2]
Выдать таблицу (билеты в наличии): рейс - класс - кол-во мест

Дано: дата d, маршрут [Z1, Z2], рейс r
Выдать таблицу (свободные места): номер места m - класс - цена

Дано: дата d, маршрут [Z1, Z2], рейс r
Выдать таблицу (список пассажиров):
место - Name - выкуплено/забронировано

Дано: дата d, маршрут [Z1, Z2], рейс r, номер места m, имя пассажира Name
Забронировать билет.
Выдать билет.

Дано: дата d, маршрут [Z1, Z2], рейс r, номера мест [m1, m2...], имена пассажиров Name1, Name2 ...
Забронировать билеты.
Выдать билеты.
24 дек 04, 10:58 [1206843] Ответить | Цитировать Сообщить модератору

Re: Объектные СУБД от Versant Corporation (FastObjects .NET, FastObjects t7, Developer Suite)
SergSuper
Member

Откуда: SPb
Сообщений: 3611 Ну вот быстро сляпал. Я правда не понял чем отличается маршрут от рейса и эти понятия объеденил. Также не стал конкретизировать каждое место, просто учитывается их количество. В принципе это все можно, но появится еще пара таблиц, и дольше делать, и наглядность не улучшится. За правильность так же не ручаюсь, возможны ошибки. Но в остальном этим уже можно пользоваться :)
Структура примерно такая

```
        Race рейсы
  race ключ
  caption описание

Shedul расписание рейсов
  race  рейс
  date дата
  fly самолёт

Fly самолёты
  fly ключ
```

caption описание

Place места в самолёте
fly самолёт
class тип места (класс)
cnt количество

Sold проданные или забронированные билеты
race рейс
date дата
class тип места
cnt количество
fio ФИО
flag флаг забронирован (0) или продан (1)
price цена продажи

Теперь задачи:

Дано: дата d, маршрут [Z1, Z2]
Выдать таблицу (билеты в наличии): рейс - класс - кол-во мест
```
select race, p.class, p.cnt-sum(isnull(cnt,0))
  from Shedul s
  inner join Place p where s.fly=p.fly
  left join Sold d on d.race=s.race and d.date='нужная дата' and d.class=p.class  and d.flag=1
  where s.date='нужная дата' and s.race='нужный рейс'
  group by race, p.class, p.cnt
```

Дано: дата d, маршрут [Z1, Z2], рейс r
Выдать таблицу (свободные места): номер места m - класс - ценаselect race, p.class, p.cnt-sum(isnull(cnt,0))
```
  from Shedul s
  inner join Place p where s.fly=p.fly
  left join Sold d on d.race=s.race and d.date='нужная дата' and d.class=p.class
  where s.date='нужная дата' and s.race='нужный рейс'
  group by race, p.class, p.cnt
```

Дано: дата d, маршрут [Z1, Z2], рейс r
Выдать таблицу (список пассажиров):
место - Name - выкуплено/забронированоselect race, p.class, s.fio, s.cnt, case flag=0 then 'забронирован' else 'продан' end
```
  from Shedul s
  inner join Place p where s.fly=p.fly
  inner join Sold d on d.race=s.race and d.date='нужная дата' and d.class=p.class
  where s.date='нужная дата' and s.race='нужный рейс'
```

Дано: дата d, маршрут [Z1, Z2], рейс r, номер места m, имя пассажира Name
Забронировать билет.insert Sold

```
  select 'рейс','дата','тип места','количество','ФИО', 0, 0
```

```
Выдать билет.update Sold
set flag=1, price='цена продажи'
where race='рейс' and
  date='дата' and
  class='тип места' and
  cnt='количество' and
  fio='ФИО'
```

По идее нужные еще порверки на наличии, но это также как в первом запросе.

Ну а теперь покажите мне ОО модель и как быстро и качественно на ней можно чего-то сделать.

PS. Я понимаю что предложенное мною решение имеет основания для критики - но прежде чем критиковать выложите своё. Еще никто этого не делал (для ООСУДБ). И я слабо надеюсь что сделает.
24 дек 04, 15:16 [1208189] Ответить | Цитировать Сообщить модератору

Вариант 2. c127 Guest
Вот решения на СКЛ-е для сайбейз АСА. Могут быть небольшие ошибки, я запросы не прогонял. Кроме того я использовал схему БД предложенную SergSuper-ом, которую я возможно не совсем понимаю, тут тоже могут быть непринципиальные неточности. Например я предположил что тип самолета хранится в Fly.caption, но если он во Fly.fly, т.е. если Fly.fly это строка вида 'TU 154', то таблицу Fly в запросы можно не включать, а тип взять из Shedul.fly.
Задача 1) типы самолетов, которыми летал Scott Tiger, 1,9,12,13,18,22,28 января 2003 годаselect f.caption
from Fly f, Shedul h, Sold s
where s.race=h.race // связываются таблицы
  and h.fly=f.fly // связываются таблицы
  and s.fio='Tiger Scott'
and s.date in
('2003.01.01','2003.01.09','2003.01.12','2003.01.13','2003.01.18','2003.01.22','2003.01.28')

Задача 2) номера рейсов которые были в январе 2003 года и на которые оставалось от 10 до 20 непроданных билетов включительно, но самолет не "TU 154"
Ваше решение если не ошибаюсь не учитывает разницу между забронированными и выкупленными билетами. В невыкупленные это непроданные, денег-то нет.

```
select s.race
  from Sold s, Shadule h, Fly f
  where s.race=h.race // связываются таблицы
    and h.fly=f.fly       // связываются таблицы
    and f.caption<>'TU 154'
    and datediff(month,'2003.01.01',h.date)=0 // можно заменить на h.date between  '2003.01.01'
and '2003.01.31'
    and s.flag=1
  group by s.race
  having (((select sum(cnt) from place p where p.fly=f.fly) - count(*)) between 10 and 20)
```

2 мар 06, 04:19 [2406559] Ответить | Цитировать Сообщить модератору
Решение на К3
Структура базы по JSON

```
Fly
{
        “Type”:  “TU134”,
        “Class1placesNum”: 10,
        “Class2placesNum”: 120,
        "FlyportAddress": {
            "streetAddress": "Московское ш., 101, кв.101",
            "city": "Ленинград",
            "postalCode": 101101
          }
}

Race
{
        “Fly”:  Fly:UID,
        “Date”: ’12-10-2001’,
        “Where”: ‘Anapa’,
        “Tickets”: [Ticket:UID]
}

Ticket
{
        “Race”:  Race:UID,
        “Class”: 1,
        “Count”: 3,
        “FIO”: “Petrovich A.A.”,
        “Flag”: 0
}
```

Задача 1 (Задача 2 и 3 и с127.1 аналогичны).
Посчитать невыкупленные билеты на заданную дату и маршрут.
E3Race. ForEachKeyRange(“Date”, Functor1, “Novosib”, “10-10-2001”, “10-10-2001”);

```
Bool Functor1(k3obj* obj, void* param)
{
        K3Race * objRace = (K3Race*) obj;
        Const Wchar_t *scity = (LPCSTR) param;
        If (objRace->Where == scity)
        {
                Int Sum1Class = Sum2Class = 0;

                For(objRace ->Ticket::Iterator itt = objRace ->Tickets.begin(); itt != objRace
        .Tickets->end();  itt++)
                {
                        K3Ticket objTicket;
                        E3Ticket.Load(*itt, objTicket) ;
                        if( objTicket.Class = 1 && objTicket.Flag != 1) Sum1Class++;
                        if( objTicket.Class = 2 && objTicket.Flag != 1) Sum2Class++;
                }
```

```
                K3Fly objFly;
                E3Fly.Load(objRace->Fly, objFly);
                cout << objRace->UID << “  Fly:” << objFly.Type ;
                cout  << “  Class1: ” << objFly.Class1placesNum - Sum1Class;
                cout  << “  Class2: ” << objFly.Class2placesNum - Sum2Class;
        }
}
```

Задача 4
Забронировать и выдать билет

```
E3Race.Load(UID,  objRace);
objTicket = new Ticket(objRace.UID, ’15-02-2000’, 1, 3, “Pupkin V.C.”, 0);
objRace.Tickets.Insert(objTicket);
E3Ticket.Save(objTicket);
E3Race.Save( objRace);
objTicket.Flag = 1; // выдать билет
E3Ticket.Save(objTicket);
```

Задача c127.2
Номера рейсов, которые были в январе 2003 года, и на которые оставалось от 10 до 20 непроданных билетов включительно, но самолет не "TU 154"

```
E3Race. ForEachKeyRange(“Date”,  Functor1,  "TU 154", “01-01-2003”, “31-01-2003”);

Bool Functor1(k3obj* obj, void* param)
{
        K3Race * objRace = (K3Race*) obj;
        Const Wchar_t *sType = (LPCSTR) param;
        K3Fly objFly;
        E3Fly.Load(objRace->Fly, objFly);
        If (objFly.Type != sType)
        {
                E3Ticket.Load(“Race=objRace”, lstTicket);
                Int countSale = 0;
                countSale  = count_if(lstTicket, FunctorCnt );
                Int countFree = objFly.Class1placesNum + objFly.Class2placesNum - countSale;
                If (countFree <= 10 && countFree >= 20)
                        Cout << objRace.UID;
        }
}

Bool FunctorCnt(UID uid)
{
        K3Ticket objTicket;
        E3Ticket.Load(uid, objTicket);
        Return objTicket.Flag == 1;
}
```

Элементы OLAP.Пример

Структура базы – аналог из статьи про «снежинки» в Википедии.
База продаж оригинальная

```
Sale
{
        Date: “15-10-2000”,
        ShopUID,
        Products: [{ProductType: “Boots”, ProductName : “Rieker N4667”, SalePrice: 234}, …],
        SumSale: 234
}

Shop
{
        Country: “England”,
        Address: {
             "streetAddress": "Московское ш., 101, кв.101",
             "city": "Ленинград",
             "postalCode": 101101
           }
}

SaleIndexShop
{
        SaleUID,
        ShopUID,  // primary key
}

SaleDependCounryAggregate
{
        Country: “France”,
        ProductByType: [{ProductType: “Shirt”, AverageSale: 150},…]
}

SaleIndexDate
{
        SaleUID,
        Date,  // primary key
        SumSale
}

SaleIndexMonthAggregate // depends on SaleIndexDate
{
        Year: 2001,
        Month: 12,
        SumSales: 239876
}

SaleIndexYearAggregate  // depends on SaleIndexMonthAggregate
{
        Year: 2001,
        SumSales: 1239876
}
```