БАЗЫ ДАННЫХ

Лекция 3

ПЛАН НА СЕГОДНЯ

- Схема работы типовых запросов
- Транзакции с точки зрения durability
- Теория вокруг WAL

ОБЩАЯ СХЕМА ИСПОЛНЕНИЯ

- Получаем в текстовом виде
- Синтаксический разбор, пока без семантических привязок
- Получаем AST - abstract syntax tree
- Преобразуем AST в дерево выражения реляционной алгебры

ОБЩАЯ СХЕМА ИСПОЛНЕНИЯ

- Работает планировщик
- Получаем дерево плана исполнения
- Исполняем

О ЧЕМ РЕЧЬ

- Любой запрос состоит из базовых операций реляционной алгебры
- Один запрос можно выполнять по-разному
- Но любой вариант раскладывается в базовые операции
- В конечном итоге стоит задача планирования запроса
- Сегодня разбираемся с базовыми операциями

ОПЕРАЦИИ РЕЛЯЦИОННОЙ АЛГЕБРЫ

- Проекция, выборка
- Дедупликация
- Группировка
- "Join"

МОДЕЛЬ СЛОЖНОСТИ

- Важно количество блоковых чтений с диска
- Ассимптотика в памяти не так важна
- Квадрат скорее не пугает
- Разве что на больших n

ПРЕДВАРИТЕЛЬНЫЕ СООБРАЖЕНИЯ

- Проекции неинтересны
- Выборка неинтересна без индексов
- А индексы пока не рассматриваем
- Без одного Scan-а точно не обойтись
- Стремимся минимизировать их количество

ПАМЯТЬ

- Читаем блоками, разумными для сканирующего чтения
- Количество таких блоков обозначаем M
- При операции над одной таблицей можем использовать 1 блок для чтения
- И M-1 - для хранения каких-то данных

ДЕДУПЛИКАЦИЯ

- Если таблица отсортирована по нужному ключу - все тривиально
- Если не отсортирована, многое зависит от количества записей
- И от разнообразия значений
- Хотелось бы уложиться в один проход

ДЕДУПЛИКАЦИЯ

- Заведем hash-таблицу и в ней будем хранить увиденные ключи
- На это у нас M-1 блоков памяти
- Хватит или нет - наперед не знаем
- Но можем оценить

КАК ОЦЕНИТЬ

- Общее количество записей
- Заданные ограничения, контекст
- Гистограммы
- Скетчи: HyperLogLog, например

А ЕСЛИ ОШИБЕМСЯ ?

- Сильно - не должны
- А если все-таки - часть ключей храним на диске
- Можно использовать Bloom-фильтры

ГРУППИРОВКА

- Детали могут зависеть от агрегации
- В целом - близко к дедупликации
- Только храним агрегированное значение

JOIN

- Выберем меньшую таблицу
- Зачитаем ее в M - 1 блоков
- Организуем в хеш-подобную структуру
- Другую таблицу читаем постепенно через оставшийся блок

NESTED LOOP

- Похоже на однопроходный
- Только надо несколько чтений второго файла сделать
- Загружаем в M - 1 сколько влезает блоков меньшей таблицы
- Другую таблицу читаем постепенно через оставшийся блок
- Повторяем со следующей порцией блоков

АНАЛИЗ NESTED LOOP

- B(R), B(S) - размеры таблиц в блоках
- Внешний цикл повторяется B(S) / (M-1) раз
- На каждой итерации читаем M - 1 блок S
- И B(R) блок R

АНАЛИЗ NESTED LOOP

- Получаем: (B(R) + M - 1) B(S) / (M - 1)
- Сокращаем: B(S) + B(S)B(R) / (M - 1)
- Плохо, если обе таблицы большие
- А если S не сильно больше, чем M, то более или менее B(R) / (M - 1)

СОРТИРОВКА

- Многие алгоритмы лучше идут на сортированных данных
- Есть модификация merge-sort под внешние данные
- Зачитаем в M блоков данные таблицы, отсортируем в памяти
- Запишем на диск
- Будем повторять, пока не закончатся данные

СОРТИРОВКА

- Получили B(S) / M отсортированных кусков
- Если их меньше, чем M, то можем слить за один проход
- Если больше, нужна еще итерация
- Но это экзотично много

СОРТИРОВКА

- Оценим сложность в блочных операциях
- Последнюю запись не считаем, в силу конвейерного контекста
- Если B(R) < M, то B(R)
- Если B(R) < M(M - 1), то 3B(R)
- Иначе - 5B(R)+

ПРИВЯЖЕМСЯ К РЕАЛЬНОСТИ

- Пусть не жалко 2G на исполнение запросов
- Есть лимит на 20 параллельных сеансов
- По 100Mb на сеанс
- Пусть блок - 1Mb, M = 100

ПРИВЯЖЕМСЯ К РЕАЛЬНОСТИ

- С таблицей До 100Mb уложимся в один проход (B)
- От 100Mb до 10G уложимся в два прохода (3B)
- От 10G до 100Tb в три прохода (5B)
- Квадрат при M дает важное следствие

ПРИВЯЖЕМСЯ К РЕАЛЬНОСТИ

- Памяти столько же, но блок 100Kb, M = 1000
- С таблицей До 100Mb уложимся в один проход (B)
- От 100Mb до 100Gb в два прохода (3B)
- От 100Mb до 100Pb в три прохода (5B)
- Но рискует потерять время на merge-е - trade-off

ДЕДУПЛИКАЦИЯ ЧЕРЕЗ СОРТИРОВКУ

- Делаем первый проход сортировки
- На втором проходе - тривиально дедуплицируем
- По ходу можем отсортированное сохранить
- Если статистика запросов располагает
- Группировка - аналогично

JOIN: ИДЕЯ И ПРОБЛЕМА

- Идея: вместо глобального вложенного цикла - локальные
- По совпадающим ключам
- Проблема: локальные циклы могут вырождать в глобальные
- Или "почти глобальные"

ПЕРВЫЙ ВАРИАНТ

- Сортируем обе таблицы
- Читаем в merge-режиме обе
- Каждой по блоку, M - 2 блоков свободны
- Ищем момент, когда ключи совпадут

ПЕРВЫЙ ВАРИАНТ

- Если совпали, делаем местный Nested Loop
- Если записей с таким ключем "совсем немного" в обоих
  - Пробегаем циклом "прямо на месте"
  - Идем к следующим ключам в обоих

ПЕРВЫЙ ВАРИАНТ

- Если только в одной "совсем немного"
  - Тянем значения из другой
  - Организуем циклический перебор
  - В конце перемещаем первую на следующий ключ
  - А вторая - уже там

ПЕРВЫЙ ВАРИАНТ

- Если в обоих "конца не видно"
  - Организуем Nested Loop
  - Задействуем M - 2 блока
  - Нужно оценить, у кого записей меньше
  - Или размер меньше

В ОБОИХ "КОНЦА НЕ ВИДНО"

- Если меньший влезает в M - 2 блоков, то ничего не теряем
- Если не влезает - приходится делать дополнительные обмены
- Утешение 1: в контексте общего планирования этого можно избежать
- Утешение 2: Это скорее всего плохой запрос

ОЦЕНИМ СЛОЖНОСТЬ

- Два прохода на сортировку, один на join - 5 (B(R) + B(S))
- Плюс возможные издержки на "толстые склейки"
- Требования: B(R) < M^2, B(S) < M^2
- Можем воспользоваться уже отсортированным
- Можем сохранить отсортированное

ВТОРОЙ ВАРИАНТ

- Не будем завершать сортировки
- Сделаем первую фазу для обоих таблиц
- И начнем действовать, как будто хотели обе слить
- Но будем организовывать циклы по совпадающим ключам

ПЛЮСЫ И МИНУСЫ

- Экономим количество проходов
- Но меньше гибкости при "толстых склейках"
- Нет этих (M - 2) блоков
- А в них сидят данные из других кусков

АНАЛИЗ

- Два прохода (3 (B(R) + B(S))
- Требования: (B(R) + B(S)) < M ^ 2
- Отсортированных таблиц не оставляем
- Отсортированными воспользоваться можем - но зачем ?

КАК ВЫБРАТЬ ?

- Можно использовать Bloom Filter или Counting BF
- Скетчи, гистограммы, ограничения, контекст
- Иногда это может привести к сильной оптимизации
- Если локализуем точку реального join-а

АЛГОРИТМЫ НА ХЕШАХ

- Пройдем по таблице, читая через один блок
- M - 1 блоков - выделим по хеш-значению на блок
- Для каждой записи посчитаем хеш и поместим в соответвующий блок
- По мере заполнения блока пишем на диск

АЛГОРИТМЫ НА ХЕШАХ

- В итоге порежем данные на M - 1 частей
- В каждой - записи с одним хешом
- Профит для дедупликации - все дубликаты в одной части
- Профит для группировки - записи с общим ключом группировки в одной части

АЛГОРИТМЫ НА ХЕШАХ

- Если B(R) < M^2, то есть шанс получить однопроходные группировки
- Есть нюансы с неравномерными распределениями ключей хеширования
- Редкие "хвосты" будут занимать блоки: можно побороться, но не так критично
- Можно получить длинные части

УДЛИНЕНИЯ

- В этом ничего нет плохого - скорее всего
- Зло в разнообразии значений
- А длинные они скорее всего из-за повторяющихся ключей

HASHJOIN

- Берем и расшиваем обе таблицы по хешам
- И делаем JOIN для кусочков с общим значение
- В среднем есть шанс получить M однопроходных
- Если количество записей в каждой - в пределах M ^ 2

НЕПРИЯТНАЯ СИТУАЦИЯ

- Два куска для одного кеша оказались большими
- Если в обоих таблицах есть доминирующие значения
- Осталась необработанной большая часть таблицы
- Однопроходный Nested Loop не годится

КАК БЫТЬ

- В любом случае можно пойти на SortJoin
- Можно из контескта понять, что для этой части JOIN неактуален
- Или из скетчей
- Можно на всякий случай сортировать куски при хешировании

РАБОТА С ПАДЕНИЯМИ

- Это часть работы с транзакциями
- Начнем с общей теории
- Введем формализованную схему описания транзакции
- Поймем, чего хотим
- Сформулируем варианты

ОПИСАНИЕ ТРАНЗАКЦИИ

- INPUT(X) - копирование с диска в память
- X может быть записью или блоком
- READ(X, t) - создаем локальную копию внутри транзакции
- t := t + 5 - изменение локальной переменной

ОПИСАНИЕ ТРАНЗАКЦИИ

- t := t + 5 - изменение локальной переменной
- WRITE(X, t) - перенос состояния в область общей видимости
- OUTPUT(X) - запись на диск

ПРИМЕР

Action	t	Mem A	Mem B	Disk A	Disk B
`READ(A,t)`	8	8		8	8
`t := t*2`	16	8		8	8
`WRITE(A,t)`	16	16		8	8
`READ(B,t)`	8	16	8	8	8
`t := t*2`	16	16	8	8	8
`WRITE(B,t)`	16	16	16	8	8
`OUTPUT(A)`	16	16	16	16	8
`OUTPUT(B)`	16	16	16	16	16

ПРОБЛЕМА АВАРИЙНОГО ЗАВЕРШЕНИЯ

- Хотим, чтобы при внезапном рестарте транзакция не была "в середине"
- Либо оба умножились, либо никто не умножился
- Будем вести лог
- Это можно делать разными способами

ОБЩИЕ МОМЕНТЫ

- Заводим лог-файл
- Заносим туда записи в режиме добавления и делаем flush
- В начале транзакции пишем
- В конце пишем

ОБЩИЕ МОМЕНТЫ

- Между START и COMMIT могут быть другие записи
- Записи, соответствующие транзакции, помечаются id транзакции
- Транзакции могут перемежаться
- Нас пока не волнуют проблемы версионирования, конфликтов

ОБЩИЕ МОМЕНТЫ

- Нас волнует восстановление
- Но в момент аварии могут жить несколько транзакций
- Время жизни транзакции не обязано быть "коротким"

COMMIT

- Конкретная семантика COMMIT зависит от метода
- В целом означает "все сохранено, ничто не пропадет"
- Но не факт, что все лежит прямо в штатной таблице
- Если нет, то гарантируется, что при восстановлении оно туда попадет

COMMIT

- Конкретная семантика COMMIT зависит от метода
- В целом означает "все сохранено, ничто не пропадет"
- Но не факт, что все лежит прямо в штатной таблице
- Если нет, то гарантируется, что при восстановлении оно туда попадет

UNDO LOG

- Перед OUTPUT пишем в лог запись типа (T, A, value)
- T - транзакция, A - переменная, value - значение
- Перед COMMIT все измененные переменные обязаны быть записаны на диск

ПРИМЕР

Step	Action	t	M-A	M-B	D-A	D-B	Log
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<$T, A, 8$>
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<$T, B, 8$>
8)	FLUSH LOG						
9)	OUTPUT(A)	16	16	16	16	8	
10)	OUTPUT(B)	16	16	16	16	16	
11)							<COMMIT T>
12)	FLUSH LOG						

СХЕМА ВОССТАНОВЛЕНИЯ

- Сканируем лог с конца
- Отмечаем встреченные COMMIT-ы
- Встретив запись об изменении, смотрим на транзакцию
- Если не закрытая, то восстанавливаем значение
- Пишем в конец лога маркер <ABORT t>