

Базы данных

Миша

24 декабря 2017 г.

Лекция 1

Преподаватель: Дмитрий Барашев

Организационное

Практос: реляционные БД(SQL)

Лектос: теория БД и внутреннее устройство.

Оценки.

По практике можно заработать 0 или -1 балл(по сути это зачет и незачет)

На лекциях можно заработать [0 ... 5]. Экзамен. Письменный. Будут какие-то задачи. Оценки суммируются. Получить 5 баллов очень сложно. Со слов преподавателя.

Список литературы.

Крис Дейт “Введение в базы данных” теоритические аспекты

Джефри Ульман, Г. Гарсиа-Молина, Уидом “Базы данных полный курс” железо, алгоритмы

Википедии доверять стоит не всегда.

Введение

Что хотим с данными делать:

- уметь обрабатывать
- поиск сортировка

Хотим добавить заголовок. Смысл.

Метаинформация:

- тип данных.
- название,
- ограничения

Определение БД можно сформулировать следующим образом: интегрированное самодокументированное хранилище информации.

Но поговорим лучше о применении БД. Мы будем использовать PostgreSQL.

С точки зрения данных:

Это набор файлов(а может и нет). Они находятся в некоторой единой структуре(физическая организация)

С точки зрения кода:

prog.exe программа, управляющая данными. Обычно это называют СУБД.

Модели данных:

- Объектно-Ориентированная
- Традиционная(таблички)
- Иерархические(JSON) и как продолжение этой модели — графовая

Что делает СУБД?

1. Управляет размещением данных на диске
2. Выполняет запросы
3. Предоставить высокоуровневый язык для манипуляции с БД
OQL(object query language), SQL, MongoDB(монголибашний язык)

Нельзя просто так взять и перенести из одной БД в другую данные на SQL.

Стандарты есть, но всем пофиг

4. Разделение прав доступа.(управление ролями). Представление данных для разных ролей
5. Инструменты для бэкапов
6. Управление конкурентным доступом
7. Восстановление после сбоев

Т.е. СУБД делает много. Какие виды СУБД?

Архитектура информационных систем с БД

1. Встроенные БД.Например SQLite
2. Клиент-серверная архитектура. Но есть проблема: очень сложно управлять. то есть допустим заведуем БД в городе и чтобы реализовать какую-то фичу, придется обновится клиента на всех клиентских компьютерах. Если в БД используется где-нибудь в госструктурах, то можно стреляться. Даже если есть автообновление, деятельность ВСЕЙ БД приходится на какое-то время останавливаться.

Поэтому люди задумались, чтобы код SQL жил где-нибудь поближе к ядру СУБД.

3. Многоуровневая архитектура. то есть СУБД <=> сервер приложений и уже к этому серверу направляется пользовательский ввод.

Поэтому в этом случае мы просто обновим сервер приложений. Эта система доминирует на данный момент в Web-приложениях.

Практика

Цель: получение некоторого законченного проекта.

Виды реляционных СУБД:

1. Oracle
2. IBM DB2
3. MS SQL Server

4. MySQL / MariaDB

5. PostgreSQL

Почта: dmitry@barashev.net
sqlitebrowser — простой GUI для sqlite

Нужно разбиться на команды по 3-4 человека и делать задания самостоятельно. Задания по практике будут командными. Но за это не будут даваться баллы. Зачёт/незачёт будет ставится по контрольным, которые будут уже индивидуальными. Поэтому общаться с друг другом стоит.

К следующему занятию нужно научиться работать с PostgreSQL из терминала.

Лекция 2

Реляционная модель

В ООП есть свои кирпичики — классы. В реляционной модели это таблицы или отношения.
Основной элемент: отношение(relation)

Домен это:

:= некоторое множество значений(бесконечное или конечное)

:= тип данных + некоторые ограничения на этот тип

В домене определены некоторые операции. Сравнение, + * - /

Примеры доменов: BOOL, INT, TEXT, INT > 0 и так далее.

Рассмотрим следующую функцию:

$f : D_1 \times \dots \times D_n \rightarrow \text{true}, \text{false}$

D_i — домен

$\{D_i\}_{i=1}^n$

f — отношение на доменах D_1, \dots, D_n .

Есть и другое определение:

Отношение: схема + тело.

Схема отношения — множество атрибутов. то есть $S = (a_i, D_i)$, a_i — имя, D_i — домен. Атрибут это пара. $a_i == a_j \Rightarrow$ атрибуты равны.

Тело отношения — множество картежей. Кортеж $t = (a_i, v_i)$ a_1, \dots, a_n — последовательность имен атрибутов в S .

$v_i \in D_i$

таблица №1

Это графическое представление.

Можно также изобразить тоже самое в виде JSON:

```
{  
    "a1": 1,  
    "a2": true,  
    "a3": "abc",  
    "a4": 3.14  
}
```

Если предикат f возвращает истину, то строчка есть в таблице, иначе ее нет.
Однаковые строчки мы запрещаем.

Равенство картежей

$$t = t' \text{ если } V_{a_i}(t) = V_{a_i}(t') \forall a_i \in S$$

Чего нет:

1. Нет одинаковых картежей(на практике всем пофиг)
2. Нет упорядоченности картежей
3. Нет упорядоченности атрибутов
4. В картежах значений: атомарны

Аспекты модели данных

1. Структура
2. Операции
3. Ограничения

Простейшие ограничения.

1. Домен — ограничение множества значений
2. Ограничения на атрибут внутри одной таблицы
3. NULL или NOT NULL.

Потенциальный ключ.

Для некоторого отношения R подмножество $K \subseteq S$ называется потенциальным ключом, если известно, что в любом экземпляре(конкретный набор строчек, то есть схема с конкретным телом) этого отношения

1. Если в любом экземпляре R для t и t' из $V_k(t) = V_k(t') \Rightarrow t = t'$ разных картежей с одинаковым ключом не бывает
2. (Неизбыточность) не существует $K' \subset K$ для K' выполняется 1.

Если выполняется только 1. то это суперключ.

Рассмотрим таблицу Студенты

направление | №паспорта | ФИО | №Студ. билета | №Группы

№паспорта и № студ. билета могут рассматриваться как потенциальные ключи.

По этим штукам можно однозначно определить студента.

Суперключ: например №паспорта + ФИО

Неизбыточность нужна, чтобы моделировать правила предметной области точно. не было студентов с совпадающими номерами паспортов

Мы поговорили о потенциальный ключе (candidate key). Также существует и первичный ключ(primary key). Первичный ключ – потенциальный ключ, использующийся в качестве основного/по умолчанию.

Лекция 3

Реляционная алгебра

Модель данных включает в себя структурную часть, ограничения и операции над объектами. Алгебра, потому что похоже на σ -алгебру множеств из матана. Операции с множеством не выводят за пределы множества.

Вход операции – отношение $1+$ (от одного). Выход – отношение

Для каждой операции нужно определить схему и тело, чтобы на выходе получить отношение.

Несколько определений

Пусть R – отношение. Тогда (R) - схема R

Если $t \in R$ – кортеж

$A \subset (R)$ подмножество схемы

$t\{A\}$ - проекция кортежа. Т.е. это набор пар: $\{(a_i, v_i)\}, a \in A$

Определение операций $\cup \cap \setminus$:

$(R \cup S) = (R) = (S)$

$(R \cap S) = (R) = (S)$

$(R \setminus S) = (R) = (S)$

$\cup \cap \setminus \times \bowtie$ – бинарные операции

$\sigma \pi \rightarrow$ унарные

Первые три – теоретико-множественные.

$R \cap S = ?$, R и S должны быть совместимыми. Что это значит?

при $r \in R, s \in S$ (R) и (S) совместимы. Идеально: $(R) = (S)$ как множества пар (a_i, D_i)

Например следующее не подходит(разные атрибуты)

$R = (a \text{ INT})$

$S = (b \text{ INT})$

в этом случае: $R \cup S$ смысл имеет но $(R) \neq (S)$

$R \xrightarrow[a \rightarrow a']{} R'$ переименование атрибута

$$R = \begin{array}{c|cc} & a & b \\ \hline 1 & & 2 \\ \hline 3 & 4 \end{array}$$

$$R \xrightarrow[a \rightarrow c]{} \begin{array}{cc} c & b \\ \hline 1 & 2 \\ 3 & 4 \end{array}$$

В итоге мы пришли к следующему: (R) , (S) совместимы если переименованиями можно получить $(R') = (S')$

Размер должен быть одинаковым: $|(R)| = |(S)|$

Выборка σ

σ , SELECT, WHERE

θ - предикат на R

$\theta(t) \rightarrow \text{true, false}$

Выборка: $\sigma_\theta(R)$

$t' \in \sigma_\theta(R) \Leftrightarrow \exists t \in R : \theta(t) = \text{true}, t' = t$

схема: $(\sigma_\theta(R)) = (R)$

Выборка в SQL на самом деле это WHERE.

Чему же соответствует SELECT?

R - отношение, $A \subseteq (R)$

$t' \in \pi_A(R) \Leftrightarrow \exists t \in R t\{A\} = t'\{A\} = t'$

Схема проекции: $(\pi_A(R)) = A$, размера тела: $|R| \geq |\pi_A(R)|$

Декартово произведение $R \times S$

Формальное требование: $(R \cap (S)) = \emptyset$ - не должно быть одноименных атрибутов

Есть небольшая разница: $(1, 2) \in INT \times INT$

В реляционной модели понятие "пары" не существует. Это единственное отличие от традиционного декартова произведения.

Формально: $t \in R \times S \Leftrightarrow \exists r \in R, s \in S t\{(R)\} = r, t\{(S)\} = s$

Схема: $(R \times S) = (R) \cup (S)$

Пример двух одинаковых запросов в SQL

SELECT * FROM T, R = SELECT * FROM T CROSS JOIN R

Соединение \bowtie

$R \bowtie S$ – естественное соединение(natural join)

$(R) \cap (S) \neq \emptyset$ были общие, равные по именам и типам

схема: $(R \bowtie S) = (R) \cup (S)$

тело: $t \in R \bowtie S \Leftrightarrow \exists r \in R, s \in S : t\{(R)\} = r, t\{(S)\} = s$

$t\{(R) \cap (S)\} = r\{(R) \cap (S)\} = s\{(R) \cap (S)\}$

Логическая реализация:

```

1   $X = (R) \cap (S)$ 
2  for r in R:
3      for s in S:
4          if r[X] = s[X]:
5              (r, s) → result

```

$R \theta S$ - тета-соединение.

θ - предикат на $R \times S$

$t \in R \theta S$ если $t \in R \times S$ и $\theta(t) = true$

Наличие $(R) \cap (S) \neq \emptyset$ – неважно

Рассмотрим запрос:

SELECT * FROM R, S WHERE R.b = S.b

Испугаемся: вроде можно сделать декартово произведение(это много), но реально все внутри оптимизируется.

Размер: $|R \theta S| \leq |R| \cdot |S|$

SELECT * FROM R JOIN S ON R.b = S.b это тета-соединение

Лекция 4

Нормализация и то, что она использует

Хотим понять, какие есть критерии качества БД и есть ли такое понятие как качество.

Определение. Функциональная зависимость: $X \rightarrow Y$, если $X, Y \in (R)$

Словами это произносится так: X – функционально определяет Y , а Y функционально зависит от X

\forall экземпляра R $t_1\{X\} = t_2\{X\} \Rightarrow t_1\{Y\} = t_2\{Y\}$

$X \rightarrow Y?$ Зависимость есть

X	Y	Z
4	2	a
3	1	b
4	2	c
3	1	d

Пришел некто и выполнил запрос:

INSERT INTO R(X, Y, Z) VALUES (4, 1, e)

X	Y	Z
4	1	e
4	2	a
3	1	b
4	2	c
3	1	d

Теперь зависимости нет

Неприводимая слева ФЗ. $X \rightarrow Y$. X мы будем называть детерминантом.

Если $\nexists X' \subset X : X' \rightarrow Y$

Рассмотрим таблицу: R(K, X, Y) K - ключ

K – определяет X, Y. Просто по определению ключа.

А что насчет такого: $KX \rightarrow Y$ Она приводима слева, потому что можно выкинуть X

Аксиомы Армстронга

Применение – нахождение ключа в отношении.

Нахождение ключа

$S = FD(R)$ – множество функциональных зависимостей для R

Если мы можем взять атрибут X и доказать, что $X \in (R)$, если $X \rightarrow (R)$, то X суперключ по определению.

Если мы навыводили зависимости, то было бы круто, если БД их проверяла.

Определение. S^T – замыкание множества ФЗ множества S, т.е. применяя аксиомы Армстронга до тех пор, пока множество не перестанет расти.

Определение. S_1, S_2 два множества ФЗ. S_2 является покрытием S_1 если $S_2^T \subseteq S_1^T$

$S_1 \sim S_2$, если $S_1^T = S_2^T$
эквивалентно

Найти минимальное $S' : S'^T = S^T$

S' минимальное, если состоит из

1 $X \rightarrow A$, где A состоит из одного атрибута

2 $XY \rightarrow A$ неприводимая слева

3 Никакую ФЗ не выкинуть

Этим мы заниматься не будем, потому что вычислительно сложно сделать замыкание. Но при этом не сказать, что это слишком актуально.

Задача поиска ключа в БД уже более полезна. Т.е. является ли данный набор атрибутов ключом или нет?

$Z \subseteq (R)$ – атрибуты, $S = FD(R) - \Phi Z$

$Z^+ \subseteq (R)$ – замыкание Z над S – множество всех атрибутов, зависящих от Z

Алгоритм построения на слайде: TODO

Рассмотрим его на примере нашей космической таблицы:

X	Z ⁺
0	CD
1	CDSPR
2	CDSPR + FYL dist

CD – суперключ

Задача

Дано отношение R. В детерминантах куча всего. Хотим получить на выходе много отношения но в ФЗ стоит только ключ

Для этого у нас будет два инструмента. Первый будет оценивать качество таблицы, второй будет препарировать и делать ей ампутацию.

Нормальная форма

Критерий качества, можно воспринимать как предикат. Т.е. если условие выполняется, то говорим, что отношение находится в какой-то нормальной форме.

Будем считать, что в отношении один ключ, чтобы упростить себе жизнь.

Нормальные формы

- Первая. Любое отношение находится в нормальной форме.
- Вторая. Любая зависимость неключевого атрибута от ключа неприводима слева.
- Третья. В зависимостях неключевых атрибутов отношения детерминантой является потенциальный ключ.
- Нормальная форма Бойса-Кодда. Для любой нетривиальной ФЗ детерминантой является какой-то потенциальный ключ.

Аномалии

Известно: $CD \rightarrow SD$, $C \rightarrow R$, $P \rightarrow dist$

CD – ключ. Если нарисовать стрелочки, то видно, что ключ определяет все.

C D S P R dist

Если мы хотим добавлять например планету с расстоянием, то без рейтинга капитана и других данных никак. Поэтому это проблема. Нельзя просто так взять и добавить. Если к нам добавился новый капитан, то мы его тоже не можем сразу добавить в таблицу, пока он не полетит куда-то. Все это называется аномалиями добавления.

Если капитана уволили, то хотим удалить все записи с ним, но мы можем грохнуть и кучу вспомогательной информации. Это аномалии удаления.

Если захотели обновить рейтинг, но в этот момент в таблицу кто-то вставляет строчку со старым значением, то получим не то что хотели. Это аномалия обновления.

Понятно, что всего этого хочется избежать.

Декомпозиция

Декомпозиция – процесс взятия проекций. Цель: проекции во второй нормальной форме и по пути ничего не потерять(что бы это ни значило). Также есть третья цель: ничего не приобрести нового.

Разрежем нашу космическую таблицу следующим образом:

C	D	S	P	R	dist
---	---	---	---	---	------

Плохо, мы потеряли зависимость $C \rightarrow R$. Плохо, что мы не можем ассоциировать ранг капитана. Разрежем по-другому.

C	D	S	P	C	R	dist
---	---	---	---	---	---	------

Опять потеряли зависимость между планетой и ее расстоянием.

Чтобы декомпозиция имела смысл, нужно пользоваться некоторыми правилами.

Теорема Хита(Heath). $R = (A, B, C)$ и $A \rightarrow B$, тогда : $R_1(A, B)$ и $R_2(A, C)$ является безопасной декомпозицией или декомпозицией без потерь. И $R = R_1 \bowtie R_2$, A будет ключом в R_1 и R_2

Теорема доказывается тривиально.

Пользуясь ею, можно выносить зависимости из отношения. Поэтому разрежем правильно. Получим вторую нормальную форму:

C	D	P	S	dist	C	R
---	---	---	---	------	---	---

После этого, обнаружилось, что жизнь стала немного легче. Но осталась зависимость между планетой и расстоянием.

Разрежем так, чтобы у нас была третья нормальная форма:

C	D	P	S	P	dist	C	R
---	---	---	---	---	------	---	---

Но есть примеры, где отношение находится в третьей нормальной форме, но все еще имеет те же проблемы

$CD \rightarrow PS$ $SD \rightarrow PC$, $C \rightarrow S$. У нас два ключа: CD и SD

C	D	P	S
---	---	---	---

Формально подходит под определение третьей нормальной формы. Т.е. капитан летает всегда на одном и том же корабле. Решение декомпозицией:

C	S	C	D	P
---	---	---	---	---

Но есть отношение: $A \rightarrow B \rightarrow C$ и $A \rightarrow C$. Не всякая декомпозиция хорошая. Можно декомпозицировать: АВ и АС, АВ и ВС. Оба подходят в теорему Хита. В первом мы теряем зависимость между В и С. Таким образом второй случай лучше, хотя с точки зрения теоремы Хита они одинаковые

Лекция 5

Повторение. Нормальная форма Бойса-Кодда(НФ БК)

Определение. Для любой нетривиальной неприводимой слева ФЗ детерминантот является потенциальный ключ.

Если в R один потенциальный ключ, то НФ БК просто совпадает с III НФ.

Пример 1.

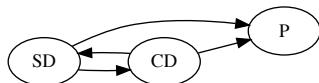
Если в отношении несколько ключей. Например: космический корабль.

S(имя), S#(номер), BirthYear, ServiceLife

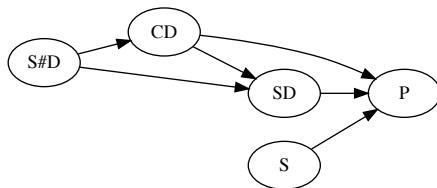
имя + номер определяют атрибуты корабля. Тут имеем НФ БК = III НФ.

Пример 2.

S, C, D, P



Вернемся к отношению с разминки:



Детерминантой является атрибут, который не является ключом. Поэтому НФ БК не подходит.

Другой пример.



$C \rightarrow S$: капитан летает на одном корабле.

$SD \rightarrow C$: для полета корабля в какую-то дату определен один капитан.

Может ли на корабле лететь другой капитан? Да, но в другую дату.

Находится ли оно в НФ БК? Ключи: SD, CD. Зависимость $C \rightarrow S$ мешает, поэтому не НФ БК.

Проведем декомпозицию.

CS, CD

По теореме Хита это декомпозиция без потерь. Все ли норм? Пропадает зависимость $SD \rightarrow C$. Печалька, потому что до НФ БК мы можем довести, но не всегда хотим, потому что по пути можем потерять нужные зависимости. Но утверждается, что такие случаи достаточно редки.

Пример.

S, BY, SL, ... (+10 атрибутов, завис. от названия корабля)

При декомпозиции возникает непреодолимое желание разбить на 10 таблиц вида:

S, BY; S, SL ...

С точки зрения денормализация(обратный процесс) дела обстоят хуже. Пользователь хочет все собранное воедино.

S, SL, D, P, Хар-ки, Dist

Нужно выполнить $n - 1$ соединение, чтобы собрать n атрибутов. А соединение это самая дорогая операция в БД.

Ссылочная целостность. Внешние ключи

Напоминание теоремы Хита. A, B, C атрибуты в схеме и $A \rightarrow B \Rightarrow AB, AC$ - декомпозиция без потерь. В отношении AB , A – ключ. И $ABC = AB \bowtie AC$

Но данные разбитые по разным таблицам должны быть как-то согласованы с друг другом. Если мы хотим добавить новый факт, то нужно записать одно и то же в обе таблицы и не ошибиться. Т.е. за этим нужно ещё следить.

Внешний ключ(Foreign Key)

Определение. Отношения R и S . K - ключ для R , $K \subset (R)$

$FK \subset (S)$ – внешний ключ, ссылающийся на K , если $K = FK$ (равны имена, типы) и в любых экземплярах R и S : $\pi_{FK}(S) \subseteq \pi_K(R)$

Основные предназначения связи между сущностями

Сущность-связь(Entity-Relationship)

Многие ко многим: N:M. Если рассмотреть множество объектов в этой ассоциации, то мы допускаем связь между любой парой.

Один к многим: 1:N. Если зафиксировать объект(капитан), то для него много полетов могут вступить с ним в ассоциацию.

Один к одному. 1:1. Взаимно-однозначное соответствие между объектами. Но надо задаваться всегда вопросом: а не один ли и тот же объект?

Пример. Капитан:Полеты. Это 1:N. Стандартно это делается через внешний ключ:

Commander(Id Primary Key) \leftarrow Flight($commander_id$ Foreign Key)

Лекция 6

Физическое выполнение запросов

Во время работы БД данные записываются в разные типы памяти. Поговорим про эти типы.

Тип памяти	Объем	скорость доступа	энергозависимость	цена/Gb
Cache процессоры	XX Mb	1ns	зависит	\$\$\$
RAM	XXX Gb	100ns	зависит	\$\$
HDD	X Tb	1-5ms	не зависит	0,0\$
SSD	XXX Gb	микросекунды	не зависит	0,\$
ленты	большой*	минуты	не зависит	?

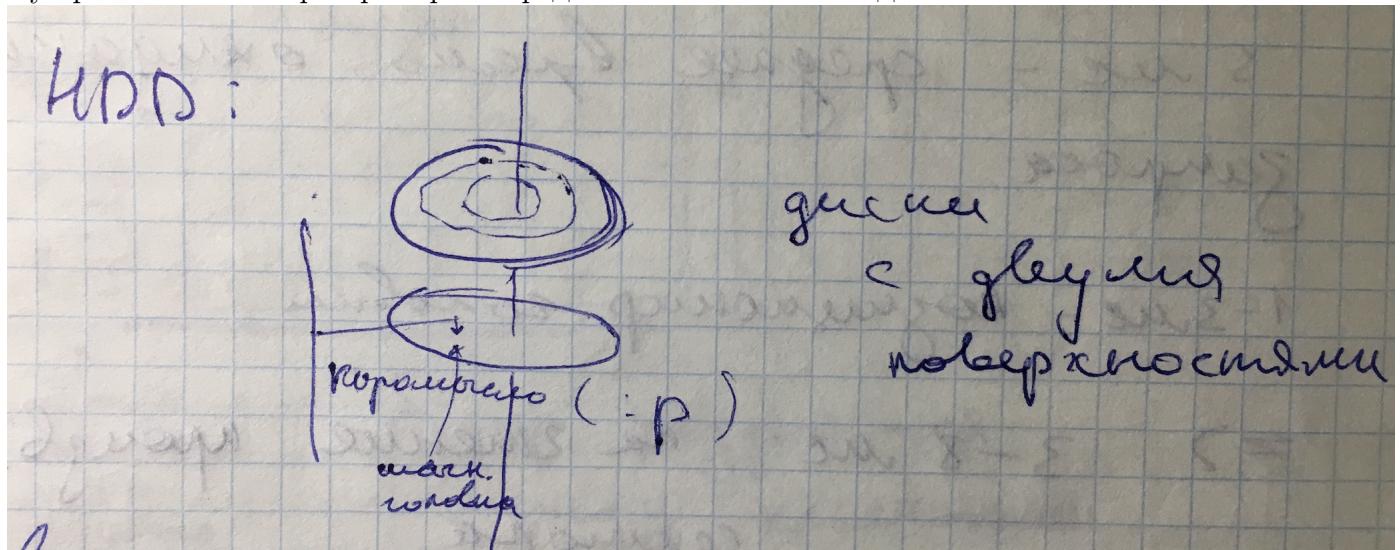
* – по сравнению с HDD

Классный сторян: RAM люди морозили в жидким азоте и если воткнуть ее в другой ноут, то данные оттуда еще можно достать. Таким образом люди перли пароль.

Ленты используются в облачном деле и используются для бэкапов.

Основная память для БД: RAM и HDD. Даже не SSD, потому что до сих пор, SSD имеет ограничение на количество циклов записи. У HDD вероятность деградирования ячеек памяти меньше. Ожидаем, что размер данных существенно больше, чем размер оперативной памяти.

Тут рассказывают про примерное представление о жестком диске.



Диск медленней оперативной памяти примерно в $10^4 - 10^5$ раз и хранит основные данные.

Multiway merge sort



Дисковая страница(несколько последовательных секторов) – единица ввода/вывода

M – размер буфера в страницах

$B(R)$ – число страниц в отношении R (сколько страниц занимает та или иная таблица)

$B(R) > M$

Пусть мы захотели отсортировать таблицу представленную в нескольких страницах.

Почему мы не хотим QuickSort? Потому что он хоть и быстрый, но при своих действиях меняет числа в массиве на разных позициях. А эти числа могут находиться в разных страницах. Это не то же самое, что просто $n \cdot \log(n)$ сравнений памяти.

I фаза. Частичная сортировка

```

1 k := [B(R)/M] + 1
2 for i in 1..k: // читаем блоки отношения k раз
3   буфер := M // блоков ⊂ R с диска
4   sort(буфер) // любая удобная сортировка
5   Ri := буфер // отсортированный список

```

II фаза. Слияние.

Обычный merge: берем два списка и мерджим их.

Multiway: то же самое, но k списков.

Реализация не слишком важна, но технически происходит следующее. Мы отсортировали какие-то блоки. Считываем в буфер первые страницы этих блоков и выбираем минимум. Минимум записываем в какую-то страницу. Когда на этой странице закончилось место – сбрасываем на диск. Повторяем процесс пока не запишем все данные.

Если $k < M \Rightarrow B(R) < M^2$

В первой фазе происходит 1 чтение и 1 запись каждой страницы. Т.е. $2B(R)$ операций.

Во второй фазе 1 чтение из R_i и 1 запись в *output*. Также $2B(R)$ операций.

Всего получается $4B(R)$ операций.

Если $M = 1Gb$, то M^2 – довольно большое пространство.

Как видно, сортировка это операция дорогая, поэтому использовать ее нужно с умом.

Пример. Размер буфера $M = 10$ блоков, списка $R = 150$ блоков. Какое количество I/O для сортировки?

Считывавем 10 блоков в буфер и в отсортированный список 10 блоков – 15 раз.

15 отсортированных списков

можем слить в один список только 10 из них.

100 блоков и 50 блоков

итого: $150 + 150 + 150 + 150 + 150 \cdot 2$

Лекция 7

Работа с блоками

Хранение отношений на диске

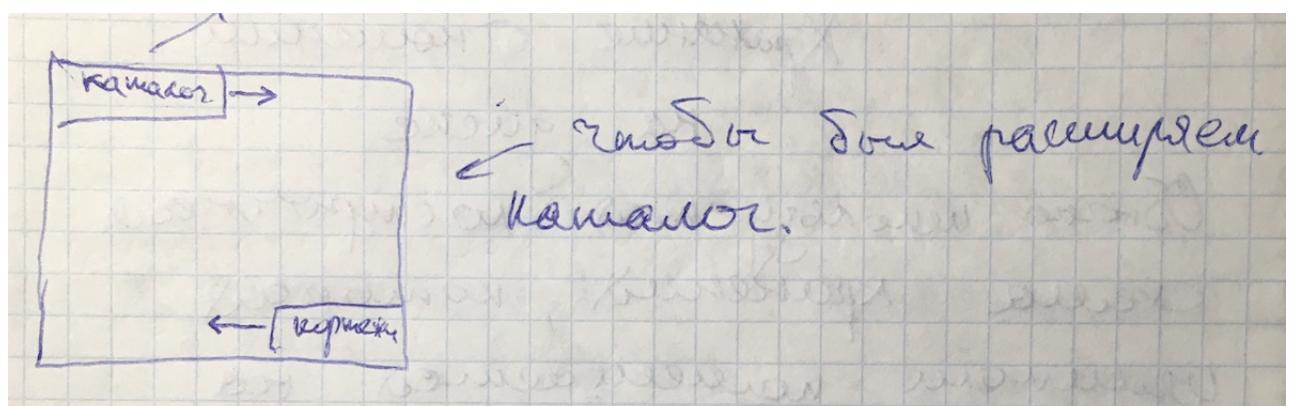
Одна страница это примерно 8кб. Как строка располагается в блоке?

Если строка состояла бы из чисел, мы бы записывали подряд.

Если мы хотим найти кортеж с номером n , то нужно получить смещение: $n \cdot \text{len(кортеж)}$

Но пусть у нас кортеж у нас не фиксированной длины. Тогда как?

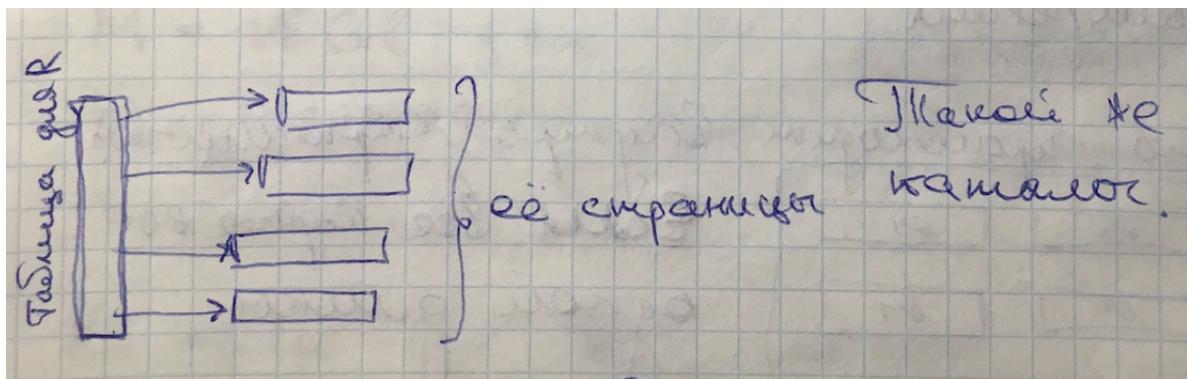
Делают таблицу ссылок. Для каждого индекса кортежа указывается область памяти, где он находится. Т.е. это указатель на память. Чтобы круче организовать память, ссылки размещаются в начале, а сами в кортеже в конце прямоугольникка.



Дальше мы фигачим связный список уже созданных страниц.

Вообще без технических подробностей, как у нас организован указатель дисковых страниц.

Поэтому таким образом, таблица отношения R разбивается на страницы.



На диске обычно хранится большой файл и СУБД читает/пишет страницы файла.

Адрес страницы: имя файла + смещение

Есть и второй вариант(архаичный?): СУБД управляет диском сама.

Адрес страницы(длинная телега): #диска #поверхности #дорожки #сектора

Отображение логического адреса в физический.

страница #42: диск1 поверхность2 дорожка8 сектор256

#43: ...

Логический указатель проходит по этим номерам страниц, из него получается физический указатель, и это уже отдается тому, кому нужно.

Колоночное хранение.

Запрос: SELECT 1, SUM(3) FROM R GROUP BY 1.

Храним атрибуты по колонкам.

+: быстрее запросы, выбираются мало столбцов

+: добавление и удаление колонок – быстро

+: "бесконечное" кол-во колонок

-: сложнее выбрать все

-: дороже и сложнее запись

Это довольно популярный способ хранения данных в современных СУБД. Чтение/запись небольшого кол-ва объектов(ограничение на странице пользователя). Читаются мало объектов, но много свойств. Это называется OLTP - online transaction processing.

Есть и другая аббревиатура. OLAP - online analytics processing. Читаются много объектов, мало атрибутов, нет операции записей(обычно редко)

online означает, что получаем ответ сразу.

Далее мы будем говорить про построчное хранение, для которого годится OLTP

Буфер и его страницы

buffer – место в оперативной памяти для дисковой страницы.

При считывании компьютер задается вопросом: нужна страница?

Если да: есть ли место в буфере? если да, то прочесть с диска в буфер.

Если в буфере нет места, то хотим найти жертву и заместить жертву нужной страницей.

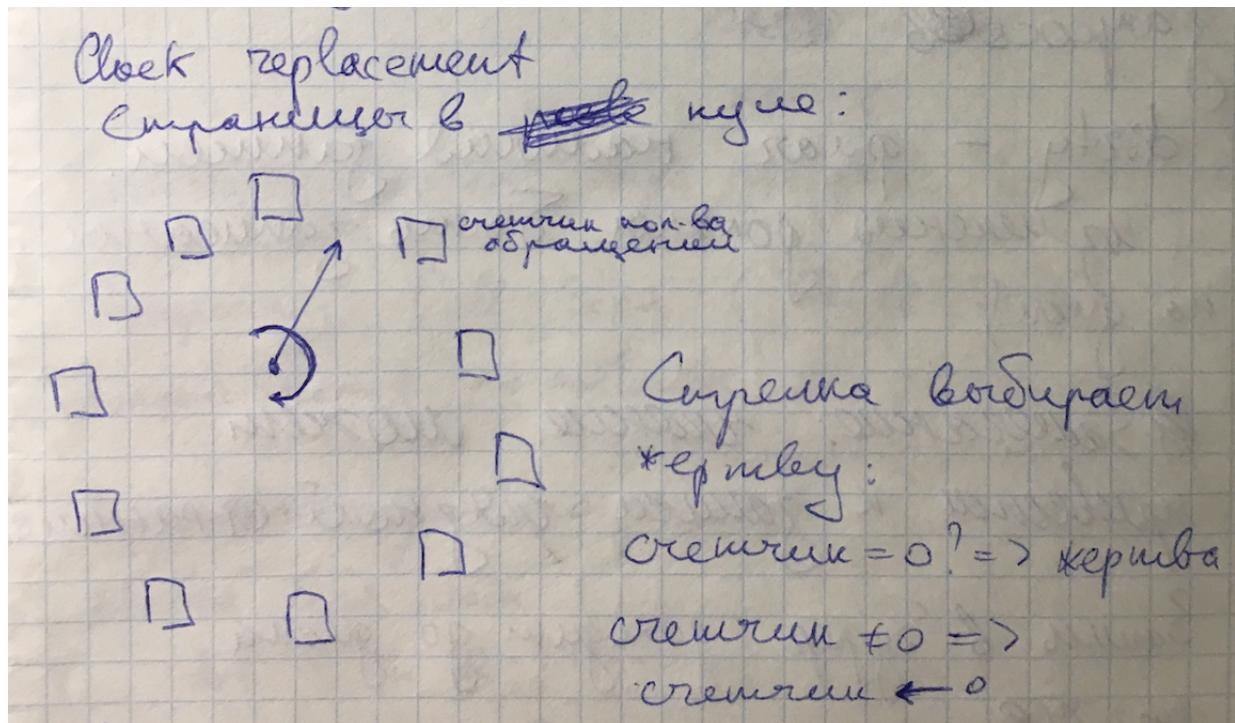
Есть несколько стратегий по нахождению жертвы.

Стратегии замещения(поиск жертвы)

- любую, т.е. случайную
- FIFO: первая в очереди.
- LRU - least recently used: та, к которой доступ был очень давно

Пользуемся последней.

Clock replacement(часовой алгоритм)



Организовали страницы на диске по кругу как часики на циферблатах.

Есть некоторая стрелка, которая вращается по часовой стрелке. У каждой страницы есть счетчик – количество обращений к ней. Двигаясь по кругу, стрелка ищет жертву по счетчикам. Если счетчик равен нулю, то эта страница нашла жертву. Если счетчик не ноль, то пока живи, но счетчик присваивается 0.

Счетчик становится 1, когда страницу читают или записывают.

Это была бинарная версия. В небинарной счетчик увеличивается на единицу при записи/чтении и действия стрелки уменьшают счетчик на единицу.

Атрибуты буферов

pin - флаг блокировки для вытеснения. Устанавливается/снимаются обработчиком запросов.
dirty - флаг наличия записей: нужно ли синхронизировать буфер с диском.

Наличие этого флага говорит:

- чтение может вызвать запись с диском
- Существует страница, запись на которую произведена, но изменения на диск еще не применились.

Задача. Пусть у нас есть некоторое отношение $R(id, value)$. 450 страниц и каталог(1 страница) с расположением остальных страниц, где располагается каталог мы знаем. У нас есть буфер размером 100 страниц. Буфер с LRU. Нет памяти кроме буфера. Выполняем запрос: `SELECT value + 1 FROM R`. Сколько раз мы прочитаем страницу каталога? Чтобы прочитать отношение, нужно сделать 5 порций чтения. И на каждой порции мы должны прочитать каталог. Чтение следующей страницы вытеснит каталог из буфера. Другой у нас ответ это 1, т.к. у нас нет памяти кроме буфера, то каталог не вытесняется из буфера по политике LRU.

Лекция 8. Алгоритмы выполнения физических операций

Нам интересна операция соединения. Алгоритм для операции выборки по какому-то условию $\sigma_\theta(R)$. Чтобы сделать выборку из таблицы, нужно ее прочитать. $B(R)$ – количество страниц в таблице. Т.е. нужно сделать сначала $B(R)$ чтений.

Первый результат: когда найдем $t : \theta(t) = true$

Сам алгоритм чтения всей таблицы называется Full Scan или Table Scan.

Nexted Loop Join

```
1 R: []
2 S: []
3 for r in R:
4     for s in S:
5         if join(r, s):
6             output ← (r, s)
```

Пусть буфер размера M т.ч. $B(R) < M$, $B(S) > M$. Чтобы эффективно организовать работу с памятью, поместим отношение R полностью в буфер и на свободное место уже будем помешать строки отношения S . Код будет в духе:

```
1 for bS in S:
2     for bR in buffered(R):
3         for r in bR:
4             for s in bS:
5                 if join(r, s):
6                     output ← (r, s)
```

Обозначим за $\text{join_blocks}(b_R, b_S)$ код с 3 по 6 строки.

Итоговая сложность: чтение($B(R)$) + чтение отношения S общим циклом($B(S)$) = $B(R + S)$. При условии того, что одно отношение маленькое, то мы можем говорить о линейном времени работы. Да, тут есть два вложенных цикла `for`, но то что происходит в оперативной памяти нас волнует меньше, чем работа с самим диском, поэтому этим мы пренебрегаем.

Теперь рассмотрим ситуацию, когда ни одно из отношений в буфере не помещается.
 $B(S) > M$, $B(R) > M$.

```

1  for i in [ $\frac{B(R)}{M}$ ]: // на самом деле в знаменателе M - 1
2      M ← chunk(R, i) // чтение одной порции R
3      for  $b_S$  in S:
4          for  $b_R$  in M:
5              join_blocks( $b_R$ ,  $b_S$ )

```

Буфер забит, но будем считать, что у нас найдется место для чтения хотя бы одного блока из S. Т.е. мы можем написать тоже самое, но M - 1. Чтобы не отвлекаться на технические детали, оставляем так.

Сложность будет: $B(R) + \frac{B(R)}{M} \cdot B(S) \sim B(R) + \frac{B(R)B(S)}{M}$

При этом, очевидно, будет без разницы что мы читаем сначала(в алгоритме читали R): R или S. Но на практике, если одно отношение больше буфера ненамного, а другое гораздо больше, то выгоднее поместить во внешний цикл то, которое меньше.

Многие отношения помещаются в буфер, поэтому использование вложенных циклов не такая уж и плохая идея.

Пример. Пусть $B(R) = 1000$. $B(S) = 500$. $M = 100$. Нашим алгоритмом мы сделаем $500 \cdot 10 + 500 = 5500$ операций. S – внешнее отношение. Почему мы не включаем в количество операции на запись результата? Во первых, можем передавать его куда-то наружу, во-вторых, запись будет добавлять константное число операций (т.е. столько же сколько и считывали с диска), поэтому не включаем.

Sort Join

$R(a, b), S(b, c)$

Если R и S отсортированы в одинаковом порядке по атрибуту b, по которому происходит соединение. Заведем два указателя на голову R и голову S. Если они равны по атрибуту b, то ура, выдаем результат и переметываем их на следующие кортежи, если неравны, то двигаем тот указатель, который меньше. Тут мы предполагали, что все атрибуты разные.

Сложность: $B(R) + B(S) = B(R + S)$

Если не отсортированы, то отсортируем через Multiway и сведем задачу к предыдущей:

$$4(B(R+S)) + B(R+S) = 5(B(R+S))$$

сортировка Sort Join

Т.е. возвращаясь к примеру, мы все сделаем за $5 \cdot (1000 + 500) = 7500$

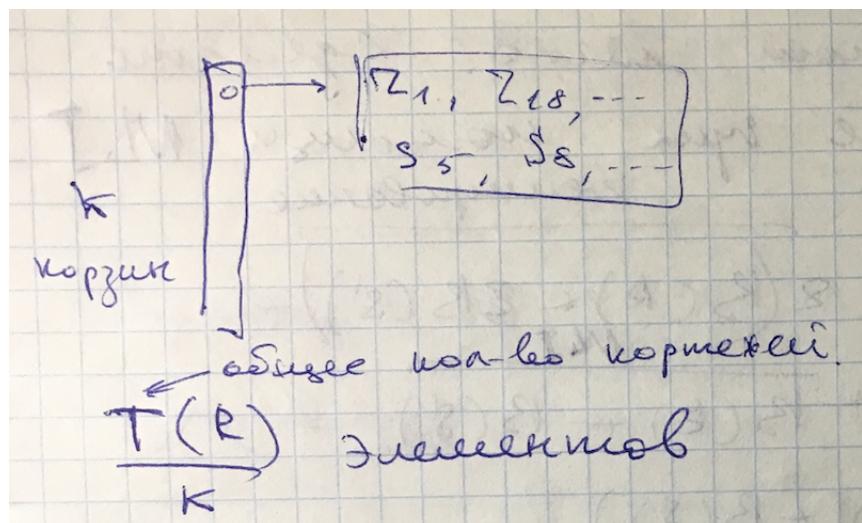
Попробуем немного соптимизировать. После первого этапа Multiway Merge Sort у нас есть отсортированные списки отношения R и S . Попробуем находить минимальный элемент в частично отсортированных списках, а в полностью отсортированном массиве. Будет работать, если отсортированных списков отношения R и S меньше чем $\frac{M}{2}$. $B(R) < \frac{M^2}{2}$, $B(S) < \frac{M^2}{2}$. Тогда сложность будет

$$3B(R + S) = \underbrace{2B(R + S)}_{\text{I стадия сортировки}} + \underbrace{B(R + S)}_{\text{слияние}}$$

Итого в примере имеем: $3 \cdot (1000 + 500) = 4500$

Hash Join

Мы хотим хешировать $R(b)$, $S(b)$, по отношению их общего атрибута b .



Если у нас будет k ячеек в хеш таблице, то в каждой ячейке будет $\frac{T(R)}{k}$ элементов, где $T(R)$ – общее количество кортежей.

Если отношение R т.ч. $B(R) < M$, то можно оптимизировать Nested Loop: искать пары по хешу. Т.е. искать для очередного $s \in S$ совпадения только в ячейке т.ч. $h(s) = h(r)$, где $h(x)$ – хеш функция.

Положим, что $\#$ ячеек = M и по этому размеру построим соответствующую хеш функцию. $h: r \in R \rightarrow [1..M]$ и $s \in S \rightarrow [1..M]$. Мы можем банально взять остаток от деления: $h = h' \bmod M + 1$, где h' – оригинальная $h': r \in R \rightarrow INT64$, но это технические детали.

Собственно, сам алгоритм:

1. Хешировать R и S по значению b . Получим на диске корзины для R и S .

Если $B(R) < M^2$ и $B(S) < M^2$ и предполагаем, что распределение элементов по ячейкам – равномерное, то тогда размер корзины/ячейки не больше M . Это мы сделаем за $2B(R + S)$.

2. В каждой ячейке сделать соединение при помощи вложенных циклов. Это сработает за линейное время: $B(R + S)$.

В итоге: $3B(R + S)$. Этот алгоритм применяется довольно часто по умолчанию.

Пример. $B(R) = 110$, $B(S) = 180$, $M = 100$. Вопрос: $R \bowtie S$ вложенными циклами. Сколько операций I/O?

Решение. Если запихнем во внешний цикл отношение из R : $100 + 180$ и $10 + 180$, сложим то получим 470. Если во внешний цикл запихнем отношение S : $100 + 110$ и $80 + 110$, сложим то получим 400.

Лекция 9. Индексные структуры

Эта структура обладает следующими свойствами:

- Вспомогательная структура
- Персистентная: индексы хранятся на том же хранилище, где и сами таблица
- Избыточная

Пример. Страница 1кб.

id	a_1	a_v
1
...
1000

Поиск без вспомогательных структур

SELECT * FROM R WHERE id=566

$E[\text{кол-во страниц при поиске по id}] = 50$

Бинарный поиск $\log_2[B(R)] = 7$

Страница индексов

Заведем страницы индексов. Т.е. для каждого id указатель на страницу.

Индекс – плотный(dense index), значит, он хранит все значения индексированных атрибутов.

8 байт на указатель(4 байта) и размер индекса(4 байта)

тогда нам понадобится 8 страниц, 125 пар на страницу.

Полный просмотр индекса + чтение из $R = 9 \text{ I/O}$

Бинарный поиск в индексе + чтение из $R = 4 \text{ I/O} = \log_2(8) + 1$

Улучшенная страница индексов

В этом случае индекс – разреженный(sparse index).

Теперь для каждой страницы мы будем хранить первый номер ее атрибута.

Теперь нам нужна всего лишь одна страница на хранение этих индексов

2 I/O: индекс + чтение из R

Имеет смысл если R отсортированно.

В каком случае это будет менее эффективно, чем плотный индекс? В случае разреженного индекса все результаты будут занимать одинаковое кол-во I/O. Если мы будем спрашивать ключ, которого нет в БД, то в плотном индексе мы сразу прочитаем индекс и увидим что нет ключа с таким значением и ответ дадим сразу.

Следующий пример.

Предположим теперь то, что строки отношения R id не отсортированы. Построим плотный индекс. Сделаем еще один слой над индексами. Сделаем разреженный слой.

Это приводит нас к следующей идее:

B-дерево(B-tree)

Мы будем говорить о модификации B+ дерева.

Описание

Элементы: дисковые страницы. Пусть мы говорим об листах.

В произвольном узле храним n ключей и n + 1 указатель.

для каждого ключа свой указатель, последний указатель нужен для указания на соседа.

Отсортированный плотный ключ.

Теперь рассмотрим слой над листьями. Тут тоже n ключей со значениями из проиндексированного атрибута и n + 1 указатель. Пусть K_i – атрибут. Указатель слева от K_i ведет на лист, т.ч. у него все ключи $< K_i$, $\geq K_i - 1$ если есть.

Теперь поговорим над общим слоем. Все тоже самое. Указатели ведут на соответствующее поддерево.
Пример дерева: TODO : картинка

Хотим, чтобы при вставке выполнялось:

- 1) Поменьше I/O
- 2) Сбалансированность
- 3) Заполненность узлов

В каждом узле листе хотим, чтобы было не менее $\frac{n+1}{2}$ пар ключ значение

В каждом узле внутренним узле не менее $\frac{n+1}{2}$ указателей.

n это максимальное количество ключей в узле.

Поиск

Если N ключей во всем дереве, то стоимость поиска это $O(\log_n(N))$

Вставка

- 1) найти лист поиском
- 2) если в листе есть место \Rightarrow записали ответ и все.
- 3) если в листе нет места, тогда нужно расщеплять лист.

Первые $\frac{n+1}{2}$ ключей остаются на прежнем месте, остальные уезжают на новую страницу. Отправляем наверх минимальное значение в этой странице: значение ключе и указатель на лист.

Пусть сверху во внутреннем узле есть место, то мы записываем указатель в нужное место и завершаем работу.

- 4) если места нет, то будет происходить деление внутреннего узла. $\frac{n}{2}$ ключей идут в левую часть, другая половина во вторую часть.

В худшем случае, мы будем расщеплять все узлы. Сложность будет также логарифмическая.

Удаление

Находим индекса и ставим на него могильный камень.

Пример. Таблица R. 10 млн строк. K = Primary key размером 4 байт. A = UNIQUE ключ размером 40 байт. Размер страницы 4k = 4000. Высота B-дерева для K и для A? Считаем, что указатель занимает 0 байт.

Решение. Если 4 байт то 1000 ключей на страницу, если 40, то 100. $\log_{1000} 10 = 3$. $\log_{100} 10 = 4$.

Лекция 10

TODO картинка

Мы хотим от ключа хранить хеш функцию. В одной корзине хранятся некоторые ключи, указывающие на соответствующие строки отношения. Точнее, это хранится в некотором блоке. Если место в этом блоке закончилось, то мы добавляем еще один блок. Теперь два блока, которые содержат ключи у которых хеш функция совпадает.

K = количество ключей

h' = h mod K

Если число ключей растет, то нужно сделать пересчитать хеш таблицу.

самый простой вариант

```
1 K = K + 2
2 foreach(key, value) in hashtable:
3     put(key, value) // кладем в какуюто_новую корзину.
```

Понятно, что это недешевая операция. Поэтому вопрос состоит в том, чтобы установить планку, по которой мы начнем все пересчитывать. Мы хотим, чтобы был только один блок в какой-то корзине. Идея: модифицировать стратегию поиска, перехешевовать только часть таблицы.

Extendible hashing(проще на вики наверн почитать)

$k = 2^i$ – размер таблицы

будем брать последние i бит от h(k)

i = 2

Картинка: пример того, как бы могла выглядеть таблица. Перебрали последние два бита и в них храним соответствующие ключи. Определение блока, куда запихнем ключ: вычисление хеш функции, берем последние два бита. Попадаем в корзину. В корзине хранятся пары (key, value). Также, в каждой корзине записано дополнительное число. Это число помогает узнать, сколько значащих бит, чтобы мы реально учитываем. Т.е. это число, сколько последних бит совпадает у всех ключей(я так понял). Это число нужно для расщепления таблицы.

пример для $i = 1$.

У нас всего две ячейки. Попадаем в корзину, в зависимости конца ключа – 0 или 1. Пусть мы вставили три ключа. 10100, 10000, 10001. Пусть у нас переполнился блок.

Пример.

Вставка:

1. найти корзину k_j , в которую мы вставим.

2. если место есть, то добавляем новое значение. иначе то может происходить разные варианты.

Случай $k_j = i$ (количество значащих битов в странице равно количеству значащих битов в таблице).

Тогда хеш таблицу надо растить.

Если $k_j < i$. Стратегия: увеличить количество значащих бит в корзине и раскидать соответственно ключи.

Было 10100, 10000. В первой корзине. Пусть добавили 10110. Теперь картинка поменялась.

Добавим еще кое что и т.к. $k_j = i$, то таблицу надо увеличить.

Объясним это на примере, когда было $i = 1$.

Добавили значение в блок, блок переполнился и надо таблицу перестроить.

Появляется вдвое больше суффиксов. Каждый ключ идет в соответствующий блок. Также увеличиваем количество значащих бит в корзинах.

Linear hashing

i - количество значащих бит в хеш-таблице.

$k < 2^i$

в корзинах допускается > 1 блок

Но мы будем следить за тем, чтобы количество ключей, деленное на количество ячеек в таблице было меньше какой-то константы: $\frac{r}{k} < C$. Т.е. ограничим среднее число ключей/корзину.

$h' = h(\text{ключ})$. Последние i bit.

Если h' – целое число = m

если $m < k \Rightarrow m$ – номер ячейки/корзины для поиска

если $m > k \Rightarrow m' = 2^i - m$

Вставка:

Если $\frac{r}{k} < C$, то все ок: добавляем просто в блок ключ.

При переполнении блока добавим новый блок.

Если $\frac{r}{k} > C \Rightarrow$ добавляем ячейку.

перемещаем из ячейки w ключи в $1w$

Если после добавление ячейки $k = 2^i \Rightarrow i = i + 1$. После этого, у нас появляется новые неиспользуемые поля.

В случае Extendible hashing места в памяти может не хватить, при перестроении таблицы в два раза.

В Linear hashing мы можем добавить произвольное количество ячеек в конец. Что делает этот способ более гибким в этом смысле.

Когда нам важен порядок – тогда хеш таблицы использовать не нужно. В-дерево про порядок знает и запрос связанный с порядком выполнит лучше.

Лекция 11

Разминка. Есть отношение $R(\text{num}, \dots)$. $T(R) = 10000$ – количество строчек в таблице, $B(R) = 100$ – количество дисковых блоков. Построено B-дерево Index (num)

TODO : картинка

Выполнен запрос: `SELECT * FROM R WHERE num > 7000.`

Кол-во I/O? Хочешь посчитать какое-то конкретное число или хотя бы указать диапазон.

Ответ. от 30 до 100. Разберемся в этом подробнее.

Какие есть возможности по использованию индексов?

Индексы бывают кластеризующие и некластеризующие.

Кластеризующий – такой, для которого выполняется свойство проиндексированного атрибута, т.е. это строки таблицы с одинаковым значением проиндексированного атрибута

Упакованы плотно, занимают минимально возможное количество блоков.

Если строки разбросаны по всей таблице, то индекс некластеризующий

В первом случае мы можем быстро считать нужные индексы.

Кластеризующий индекс по сути определяет как данные хранятся в таблице (спускаемся к грешной земле)

Но зачем тогда нужен некластеризующий?

Сколько кластеризующих индексов можем построить, если у атрибутов вообще нет зависимостей друг от друга? Один.

Оценки стоимости поиска по индексу.

$T(R)$ – количество записей/строк в таблице

$B(R)$ – количество блоков

Рассмотрим какой-нибудь атрибут. Обозначим его за a .

$V(R, a)$ – кардинальность атрибута a . Кардинальность – число различных значений этого атрибута.

Для кардинальности имеем тривиальное неравенство: $1 \leq V(R, a) \leq$ количество записей

Если a – это ключ, то верхняя оценка достигается.

Фигачим запрос по условию: $\sigma_{a=c} = c(R)$ есть индекс $R.a$

Сколько строк выберем, при условии, если значения распределены равномерно?

если значения a равномерно распределены, то тогда размер выборки $\frac{T(R)}{V(R,a)}$

количество I/O для чтения блоков?

Если индекс кластеризующий, то тогда $\frac{B(R)}{V(R,a)}$ I/O

Если индекс некластеризующий, то тогда $\frac{T(R)}{V(R,a)}$ I/O в "тупом" случае.

Но БД для начала строит map – Bitmap Index Scan, размера количества страниц.

где на i от бите хранится флагок: нужно ли доставать i -ю страницу или нет.

В запросе есть несколько условий на равенство атрибутов, то несколько bitmap и их побитовое И(быстрое), даст нужные страницы, которые нужно выдать для запроса, именно поэтому нужен bitmap.

С неравенством немного сложнее.

Гистограммы. Сколько значений хранится для каждого индекса.

Index Join

$R(X, Y) \bowtie S(Y, Z)$

для S, Y есть индекс

наивный алгоритм:

читай все строки из R

для каждой записи $r \in R$

найди s: $\{s\}_Y = \{r\}_Y$

используя индекс

все строки из R:

$B(R)$

для r соединяющихся строк $\frac{T(S)}{V(S,Y)}$

для r соединяющихся блоков $\frac{B(S)}{V(S,Y)}$

Количество I/O: $B(R) + T(R) \cdot \frac{B(S)}{V(S,Y)}$

В примерах, которые были раньше

$B(R) = 1000$

$B(S) = 500$

$T(R) = 10000$

$T(S) = 5000$

$V(S, Y) = 500$

Итого: $1000 + 10000 = 11000$, что кажется это слишком много.

Варианты улучшения

1-ый проход. $B(R)$ чтений и bitmap по индексу S.r

2-й проход: вложенные циклы

с R – внешнее отношение

блоки из S: бит в битмап = 1

внутреннее отношение

Если есть индекс для R.Y

Нужны сами записи из R и S?

Если да, то битмапы, все такое

Если нет, то Index Only Join/Scan

"Буфера памяти"

Итеративная модель вычислений

$R \bowtie \sigma_{a=c}(S)$

TODO картинка

¹ Iterator \{

```
2 Open(); // размещение структур в памяти. Подготовка к возвращению первой строки
3 результата
4 GetNext() Record; // возвращает текущую строку, переходит к следующей
5 Close(); // закрывает, убирает все из памяти
\}
```

HashJoin(Iterator left, Iterator right) implements Iterator

Физический план выполнения запроса

```
1 HashJoin \Rightarrow
2   Hash \Rightarrow
3     SeqScan
4     IndexScan(S) по условию a = c
```

Лекция 12

TODO: картинки для планов

Разминка. $R(X, Y) \bowtie X(Y, Z)$

Y - ключ в S, внешний ключ в R

Что можно сказать про $V(R, Y)$?

Ответ: оно хотя бы 0 и не больше $V(S, Y) = T(S)$

по сравнению с кардинальностью других атрибутов.

Оптимизации запросов

Оптимизатор выбирает лучший план

Например, запрос:

SELECT Z FROM R JOIN S WHERE R.x = C

Логический план:

$\Pi_Z -- \sigma_{R.x=C} -- \bowtie (R, S)$

(проекция по Z) – выборка – JOIN

Задача №1:

Применить преобразования логического плана и получить более эффективный. Эффективный – значит минимизирует/максимизирует целевую функцию. В нашем случае, она означает то, что мы хотим все сделать побыстрее. Иногда бывают и другие, например, уменьшить нагрузку.

Цель: время выполнения запроса $\rightarrow \min$

Из чего время выполнения запроса складывается, если мы хотим его минимизировать?

Мы считаем, что самая жирная операция – это работа с диском. Поэтому остальное мы игнорим (по крайней мере в этом курсе). Поэтому время пропорциональное количеству чтений/записей(I/O) с диска. Собственно, этой метрикой мы только и пользуемся.

Нарисуем эквивалентный план, где вместо соединения у нас стоит декартово произведение:

$$\Pi_Z = \sigma_{R.x=C} = \times(R, S)$$

Т.е. запрос был бы: SELECT Z FROM R WHERE R.x = C AND R.Y = S.Y

Преобразования плана с помощью алгебраических операций

Для начала вспомним их:

Коммутативность: $R \bowtie S = S \bowtie R$

Ассоциативность: $(R \cup S) \cup Z = R \cup (S \cup Z)$ – объединение

или $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$

Дистрибутивность(селекции относительно декартова произведения): $\sigma_{R.x=C}(R \bowtie S) = \sigma_{R.x=C}(R) \bowtie S$. И аналогично относительно декартова произведения:

$$\sigma_{R.x=C}(R \times S) = \sigma_{R.x=C}(R) \times S$$

Нас будет интересовать дистрибутивность.

Проталкивание предикатов(predicate push down)

Все операции, которые могут уменьшить размер отношений должны опуститься максимально глубоко. Т.е.

$$\sigma_{\theta_1 \text{AND} \theta_2}(R) = \sigma_{\theta_1}(\sigma_{\theta_2}(R))$$

Поэтому план который был выше упростится: $\Pi_z -- \bowtie (\sigma_{X=c} -- R, S)$

Можно еще немного упростить: если нам нужен только атрибут Z, то протолкнем его вниз тоже.

Кто-то может додуматься "помочь" оптимизатору:

```
1 SELECT Z FROM S JOIN (
2     SELECT * FROM R WHERE x = c
3 ) AS T;
```

В результате мы получаем менее читабельный запрос, но логический план оптимизатора остается тот же.

Хорошее быстрое выполнения запроса лучше чем Отличное.

Потому что пока мы найдем отличный вариант выполнения запроса, пользователь состарится. Поэтому над некоторыми вещами оптимизатор не заморачивается. Т.е. мы не то чтобы ищем глобальный минимум, мы даже локальный минимум можем не искать.

Вернемся к нашему плану:

$$\Pi_z -- \bowtie (\sigma_{X=c} -- R, S)$$

На месте \bowtie могу быть вложенные циклы(NL), SJ, HashJoin, что выбрать?

На месте $\sigma_{X=c}$ может быть SeqScan или IndexScan.

Что нам поможет? Оптимизатор может использовать некоторую статистику(гистограммы и тп). Сбор статистики если что тоже тратит время, что иногда тратит время работы, поэтому иногда администратор отключает это.

Селективность выборки/запроса это $\frac{1}{\text{размер результата}} / \text{размер входа}$

Применять индексы будем в случае, если селективность достаточно высока.

Выбор алгоритма соединения:

операнд слева(на дереве) < M ? Если да, то Nested Loops иначе используем HashJoin.

Немного модифицируем нашу цель. Заменим функцию стоимости: количество I/O заменяем на размер промежуточных отношений. Размер – количество строк в отношении. Что же это такое? Чем больше строк в отношении, чем больше нужно I/O. То, что у нас стоит в листьях мы не можем влияет(это то что нам дали), на результат мы тоже не влияем. То что варьируется во время вычислений это количество промежуточных вычислений.

Промежуточные результаты нужно записывать на диск, если они получаются большими(не помещаются в буфер).

В оперативной памяти находятся буфер страниц и рабочая память: там где могут находится промежуточные результаты. Рабочая память для одного клиента это около 10 мегабайт вот ее как раз запросто может и не хватить при больших результатах.

Т.е. они будут сжирать память и тратить время записи на диск.

Таким образом, мы заинтересованы в том, чтобы уменьшать размер промежуточных результатов.

Напоминание $|\sigma_{X=C}(R)| \approx \frac{T(R)}{V(R,x)}$

Если константу C мы выбираем равновероятно, то ожидание размера выборки будет одинаковым.

$E = \sum_C p_C \cdot |\sigma_{X=C}(R)| = \frac{1}{n} \cdot \sum_C |\sigma_{X=C}(R)| = \frac{1}{n} \cdot T(R)$, где n это V(R, x).

TODO: небольшой комментарий к этому

В реальности распределения значений скорее всего похожи на нормальные.

(пример гистограммы для роста)

Другой вариант: какая-нибудь вариация степенного распределения.

(пример гистограммы для размеры городов/клики на страницах)

"Сейчас будет черная магия, моя любимая"

Размер соединений

Задача: найти размер соединения.

Мы будем предполагать, что

1) Вложенность множеств значений атрибутов.

пусть у нас есть идентификаторы объектов.

Если $V(R, Y) \leq V(S, Y)$, тогда $\Pi_Y(R) \subset \Pi_Y(S)$

$R(X, Y) \bowtie S(Y, Z)$

Пусть $V(R, Y) \leq V(S, Y)$

$r \in R$ Сколько с ним соединяется? $s \in S : \frac{T(S)}{V(S,Y)}$

$|R \bowtie S| = \frac{T(R) \cdot T(S)}{V(S,Y)}$

Пусть $V(R, Y) \geq V(S, Y)$. Для $s \in S$.

$\frac{T(R)}{V(R,Y)}$ пар.

$$\text{T.e. } |R \bowtie S| = \frac{T(R) \cdot T(S)}{V(R, Y)}$$

$$\text{T.e. в итоге: } |R \bowtie S| = \frac{T(R) \cdot T(S)}{\max(V(R, Y), V(S, Y))}$$

$$\begin{aligned} Y - \text{ключ в } S (\text{Foreign Key в } R) \quad & |R \bowtie S| = \frac{T(R) \cdot T(S)}{V(S, Y)} = T(R) \cdot 1 \\ V(R, Y) = V(S, Y) = 1 \Rightarrow & |R \bowtie S| = T(R) \cdot T(S) = |R \times S| \end{aligned}$$

$$Y - \text{ключ в } R \text{ и } S, \text{ то } |R \bowtie S| = \frac{T(R) \cdot T(S)}{\max(T(R), T(S))} = \min(T(R) \cdot T(S))$$

Порядок выполнения соединений

SELECT FROM R JOIN S JOIN T JOIN U
WHERE ...

TODO: Картинки

Можем расставить скобки как нам только захочется. Соединение – ассоц. и коммут.
Леворекурсивное дерево(рекурсия влево)

Лекция 13

Разминка.

$R(a, b)$ порядок

$S(b, c)$ $R \bowtie S \bowtie T$

$T(c, a)$

	R	S	T
T	1000	5000	200
V_a	1000	—	50
V_b	2500	5000	—
V_c	—	40	200

Т это размер таблицы, кардинальность оказалась больше таблицы, но пофиг

Какой самый лучший порядок? (минимизирующий пром. результаты) Всего у нас их три:

(RS)T

(RT)S

(TS)R

Правильный второй вариант. У нас только один промежуточный результат. Нужно его минимизировать

$$|R \bowtie S| = \frac{1k \cdot 5k}{5k} = 1k$$

$$|S \bowtie T| = \frac{5k \cdot 200}{200} = 5k$$

$$|R \bowtie T| = \frac{1k \cdot 200}{1k} = 200$$

RT и дальше соед. с S

Для 3 вариантов еще несложно, а что если побольше? Будем действовать жадным алгоритмом. База пары с мин. ресурсом. То же самое для N - 1 отношения. Следующее отношение – то, которое умеет минимизирует размер соединения с текущим результатом. Т.е:
Индукция: $\min_i(R_i \bowtie (Result_{n-1}))$

Рассмотрим это на примере.

	R(a, b)	S(b, c, d)	T(c, e)	U(d, e)
T	100	100	200	200
V _a	50	—	—	—
V _b	100	100	—	—
V _c	—	2	2	—
V _d	—	2	—	20
V _e	—	2	200	200

Предположения об атрибутах

- 1) Вложенность множества значений. (мы благодаря этому построили оценку размера соединения)
- 2) Сохранение множеств значений. Т.е. если $R(a, b) \bowtie S(b, c)$ и мы считаем кардинальность V_a :
 $V_a(R(a, b) \bowtie S(b, c)) = V_a(R)$

Общая формула для соединения нескольких отношений:

$$|R_1 \bowtie \dots \bowtie R_n| = \frac{T(R_1) \times \dots \times T(R_n)}{V_{a_1,2} \times V_{a_1,3} \dots}$$

a_i атрибуты входящие в больше чем одно отношение

$V_{a_i,1}$ – наименьшая кардинальность

$V_{a_i,1} \times V_{a_i,K_i}$ – упорядоченные кардинальности общего атрибута a_i для K_i отношений

Вернемся к примеру и насчитаем кое-что:

$$|RS|: \frac{100 \cdot 100}{100} = 100$$

$$|RT|: 100 \cdot 200 = 20k$$

$$|RU|: 20k$$

$$|ST|: \frac{100 \cdot 200}{2} = 10k$$

$$|SU|: \frac{100 \cdot 200}{20} = 1k$$

$$|TU|: \frac{200 \cdot 200}{200} = 200$$

$$|(RS)T|: \frac{|RS| \cdot |T|}{2} = \frac{100 \cdot 200}{2} = 10k$$

$$|(RS)U|: \frac{|RS| \cdot |U|}{20} = \frac{100 \cdot 200}{20} = 1k$$

Получили некоторое лево рекурсивное дерево $((R, S) U), T)$

Общая стоимость $100 + 1000 = 1100$.

Получили ли мы оптимальный(мин. пром. результаты) результат? Наверное оптимальным все же было бы дерево $(R, S), (T, U)$. Или, если мы рассматриваем только леворекурсивные деревья, то лучше было бы $((T, U) S), R$. Придем к этому ответу с помощью другого алгоритма.

Динамическое программирование

Будем строить таблицу, где будем запоминать: размер результата, его стоимость получения, оптимальная последовательность для получения.

	Cost	Size
RS	—	100
RT	—	20k
RU	0	20k
ST	—	10k
SU	—	1k
TU	—	200

	Cost	Size
(RS)T	$100 = \min(RS , ST , RT)$	10k
R(TU)	200	$200 \cdot 100 = 20k$
(RS)U	100	$\frac{100 \cdot 200}{20} = 1k$
S(TU)	200	$\frac{200 \cdot 100}{2 \cdot 20} = 500$

Получили оптимальный план для каждой тройки, теперь для четверок:
Мы если что по прежнему рассматриваем только леворекурсивные деревья.

RSTU: считаем стоимости, размер должен у всех получиться одинаковым
 $(RST)U : 100 + 10k = 10100$
 $(RTU)S : 200 + 20k = 20200$
 $(RSU)T = 1100$
 $(S(TU))R = 700$

$(RS) \bowtie (TU)$ был бы оптимальнее, если бы мы рассматривали и другие деревья.

Динамика по отзывам(из postgres, где она делается по умолчанию) работает хорошо, где отношений в соединении не более чем 10-15. Если больше, то используется жадный алгоритм например.

На этом все оптимизационные штуки мы закончили.

Транзакции(Баращев сильно улыбается). Конкурентный доступ к данным.

Если есть несколько ядер, то есть возможность параллельного выполнения
Диск медленнее(чтение еще идет асинхронно) чем CPU, поэтому пока крутится диск, можно нагружать CPU еще чем-то.

Наличие прогресса выполнения запроса для пользователя.

И так, мы хотим параллельно выполнять запросы. Но с таким подходом мы получаем также следующие проблемы:

Грязное(неподтвержденное) чтение. Если какой-то запрос начал обновлять таблицу и обновляет много строк. Второй запрос читает таблицу. Где-то он получил новые данные, где-то старые. Все плохо.
Потерянное обновление. Аналогично.

Что мы ожидаем от БД, которая заявляет, то, что она поддерживает транзакции?
Транзакция – некоторая атомарная группа операций, переводящая из одного согласованного состояния в другое согласованное состояние.

Под атомарностью понимается применение/неприменение сделанных изменений. Если изменения записываются, то записываются все изменения, если не записываются, то не записывается ничего. Под согласованным состоянием понимается то состояние, где выполнены все ограничения. Какие? Такие же, которые сам программист и определил в БД: ключи, constraint и тп. Внутри самой транзакции согласованность может нарушаться и это нормально. Т.е. если мы говорим о транзакции, традиционно мы вспоминаем перевод денег между счетами. Условие: сумма на обоих счетах до и после транзакции должна совпадать. Из одной строчки мы отнимаем число. В другую строчку прибавляем это число. Между этими операциями согласованности нет.

ACID-семантика транзакций.

Гарантия, которая дает БД при использовании транзакций.

A - Atomicity. Атомарность.

C - Consistency. Согласованность.

I - Isolation. Изолированность.

D - Durability. Долговечность.

Атомарность. То что было сделано в транзакции будут выполнено, либо ни одна из операций не будет выполнена. Т.е. изменения действительно атомарны.

Согласованность. БД проверит выполнения ограничений при фиксации транзакции. Фиксация транзакции – успешное подтверждение выполнения.

Изолированность. С точки зрения программиста каждая транзакция выполняется так, как будто она в системе одна. Т.е. мы не заботимся о чтении/записи из других транзакций. Если бы мы писали на C++/Java, то привет всем Mutex и семафорам. БД берет на себя обязанность синхронизироваться между транзакциями. Это некоторая политика относительно разрешения конкурентных изменений.

Долговечность. Гарантия, что подтверждавшиеся изменения не будут случайно утеряны. Под словом случайно понимается потеря без явных действий со стороны программы. Т.е. например выключили свет после подтверждения транзакции. БД гарантирует, что изменения после восстановления питания будут.

Лекция 14

Разминка.

TODO – сорян, опоздал.

Расписание

последовательность действий транзакций.

Два вида расписаний:

Последовательное(serial) с расписанием: действия разных транзакций не перемешиваются.

Далее шел просто пример с двумя транзакциями. Операции которых, если бы перемешались была бы каша.

TODO

Сериализуемое(serializable) расписание транзакций T_1, \dots, T_n :
эффект его == эффекту некоторого последовательного расписания
Под эффектом подразумеваем состояние БД.

Рассмотрим пример расписаний:

```
1    $T_1$ :
2   READ(A)
3   A += 100
4   WRITE(A)
5
6    $T_2$ :
7   READ(A)
8   A *= 2
9   WRITE(A)
10
11   $T_1$ :
12  READ(B)
13  B += 100
14  WRITE(B)
15
16   $T_2$ :
17  READ(B)
18  B *= 2
19  WRITE(B)
```

T_1, T_2 эквивалентное последовательное расписание, значит, расписание дает такой же результат, если бы оно выполнилось ровно так как написали. T_2, T_1 – даст другой результат. Это расписанием сериализуемое. Пример посложнее:

```
1    $T_1$ :
2   READ(A)
3   A += D
4   WRITE(A)
5
6    $T_2$ :
7   READ(A)
8   A \cdot = C
9   WRITE(A)
10
11   $T_2$ :
12  READ(B)
13  B += C
14  WRITE(B)
15
16   $T_1$ :
17  READ(B)
18  B += D
19  WRITE(B)
```

Будет ли оно сериализуемом. $D = 0$ и $C = 1$, то нам все равно в каком порядке выполнять. Если D, C – случайные значения, то это расписание несериализуемое. Довольно кэпский вывод, что результат транзакции зависит от того, что в ней происходит. Предположим, что у нас нет действий программиста, остались только READ и WRITE. Сейчас нам станет жить легче.

$R_i(X)W_i(X)$

X – объект, над которым производится действия, i – номер транзакции. R, W – тип действия(Read, Write).

Перепишем пример в этой нотации:

$R_1(A)W_1(A)R_2(A)W_2(A)R_2(B)W_2(B)R_1(B)W_1(B)$

Рассмотрим понятие конфликта действий. Два действия конфликтуют, если результат расписания зависит от их порядка.

Какие действия очевидно не конфликтуют? Два чтения. В любом случае данные нужно прочитать, поэтому от их порядка ничего не зависит.

Нарисуем матрицу действий.

	$R_2(X)$	$R_2(Y)$	$W_2(X)$	$W_2(Y)$
$R_1(X)$	+	+	-	+
$R_1(Y)$	+	+	+	-
$W_1(X)$	-	+	-	+
$W_1(Y)$	+	-	+	-

Если читается и записывается одно и то же – то конфликт. Если оперируем с разными переменными – то нет(см. клетку 1 ряд 4 столбец и 4 ряд 1 столбец)

Из матрицы можем сделать вывод: если действия оперируют с одним и тем же объектом и хотя бы одна из них это запись(W), то переставить их местами мы не можем.

Что мы понимаем под объектом? Таблицу лучше не надо. С большинстве СУБД считается, что это строка.

Сериализуемость по конфликтам. Если расписание привести к последовательному перестановками рядом стоящих неконфликтных действий.(TODO пояснить немного)

Является ли пример, который был выше сериализуемым по конфликтам? Нет, поскольку есть конфликты которые мешают это сделать: $W_1(A)R_2(A)$ и $W_2(B)R_1(B)$.

Тест на сериализуемость по конфликтам

Орграф предшествования для расписания $S = T_1 \dots T_n$. Вершины – транзакции.

Ребро T_i, T_j , если в S есть конфликтующие действия транзакции T_i, T_j и действие T_i стоит перед действием T_j .

Если граф ацикличен, то расписание сериализуемо по конфликтам. И его можно получить топологической сортировкой.

Теорема. Расписание S сериализ. по конфликтам \Leftrightarrow в графе предшествования для соответствующего расписания нет циклов.

Если в графе есть циклы, то по определению.

В расписании: $T_1 \dots T_2 \dots T_3 \dots T_1$ – конфликт действий. По определению сериализуемости – ее мы не получаем.

Если в графе нет циклов. Существует вершина без входящих ребер(T_k). \Rightarrow действия T_k можно пе-

реставрировать в начало расписания (иначе в T_k было бы входящее ребро). Таким образом, можем запуститься "рекурсивно" от того, что осталось справа. Или иначе говоря, мы провели индукцию. После того как переставили T_k в начало, удалили ее из графа.

Блокировки

Планировщик на блокировках.

$R_1(X)W_2(X)$ – неповторяемое чтение

$W_2(X)R_1(X)$ – грязное чтение

$W_1(X)W_2(X)$ – потерянное обновление

На действие мы можем повесить "замочек" (mutex). Если бы каждая транзакция вешала бы mutex на каждый элемент, которая она хотела бы прочитать, то конфликты бы разрешались.

Правила (бойцовского клуба) блокировок

1. Хочешь что-то сделать с X – ставь mutex на него
2. Повесил mutex – не забудь снять (после действия)
3. Двух mutex-ов на одном объекте быть не может

Таким образом появились два новых действия: $L_i(X)$ – блокировка X i-ой транзакцией, $U_i(X)$ – разблокировка.

Эти правила можно и сформулировать с этими новыми действиями.

Если мы хотим что-то сделать с X и на нем висит mutex – то, ждем пока его снимут.

4. Все блокировки, сделанные транзакцией i предшествуют всем ее разблокировкам.

TODO график количества блокировок в зависимости от времени.

Он будет выглядеть как ступенчатый график.

Двухфазный протокол блокировок aka 2PL - two-phase locking.

Когда транзакция завершается, то блокировки все снимаются.

Поэтому у нас в конце получился обрыв. Обрыв лучше чем ступеньки, потому что есть возможность другой транзакции получить mutex и тогда будет плохой.

Теорема. Расписание, построенное с помощью с 2PL сериализуемо по конфликтам.

по индукции. База. Расписание одной транзакции сериализуемо.

Пусть у нас есть расписание из n транзакций.

Найдем транзакцию, которая первой выполняет операцию снятие блокировки. Пусть i ее номер.

Проверим, что можно получить расписание $T_i S'$, где S' все транзакции кроме T_i .

Я прослушал... здесь от противного...

Усложнение блокировок: разделяющие (shared), эксклюзивные (exclusive)

Таблица совместимости:?????

	S	X
S	+	-
X	-	-

Планировщик с временными ветками(timestamp ordering scheduler)

timestamp – монотонная возрастающая последовательность целых чисел

Системные часы(иногда дают сбой)

Генератор значений

Каждой транзакции Т

дается $TS(T)$ – логическое время ее поступления

Модель мира:

транзакции выполняются мгновенно в момент поступления

В реальности:

если расписание соответствует модели, то все ок

если расписание невозможно в модели, то злимся...

У каждого объекта X есть два timestamp и один бит информации. Т.е.

$RT(X)$ – логическое время последнего чтения

$WT(X)$ – логическое время последней записи

$C(X)$ – подтверждение записи

Две невозможные ситуации с точки зрения планировщика.

Опоздавшая запись картинка

Опоздавшее чтение картинка

$R_i(X)$

1) $WT(X) \leq TS(T_i) \Rightarrow C(X)$, то ок

$RT(X) = \max(TS(T_i), RT(X))$

если $\neg C(X) \Rightarrow$ подожди

2) $WT(X) > TS(T_i) \Rightarrow ROLLBACKT_i$ – опоздавшее чтение

$W_i(X)$

1) $RT(X) > TS(T) \Rightarrow ROLLBACKT_i$ – опоздавшая запись

2) $RT(X) < TS(T_i), WT(X) > TS(T_i)$

картинка: что делать?

$\emptyset \Rightarrow$ ничего

$\neg C(X) \Rightarrow$ либо подождать T_i либо отложить запись $W_i(X)$

3) $RT(X) < TS(T_i)$

$WT(X)$

$WT(X) := TS(T_i)$

$C(X) := False$

Многоверсионный протокол

- 1) У каждого объекта много версий
- 2) Каждая запись создает новую версию X_t
- 3) $W_i(X)$ транзакций с $TS(T_i) = t$
- 3) При $R_i(X)$ читается $X_i : t < TS(T_i)$ не существует $X_t : t' > t, t' < TS(T_i)$
- 4)

Лекция 15

Разминка.

$R_2(X)R_1(Z)W_2(Z)R_3(X)W_3(Y)W_2(X)W_1(Y)$
 $T_3T_1T_2$
???

Что может пойти не так

- 1) Ошибки оператора/человека/программы, неверное данные

решение: ограничения/проверка/святая вода/бубен

- 2) Сбой носителя

решение: синхронная запись на несколько дисков

RAID с зефирками – массив сильных и независимых дисков

- 3) Системный сбой

отключение электричества, ошибка в коде СУБД

- 4) Катастрофический сбой

зональный – сбой в dataцентре

региональный/континентальный – лоукот в энергосистеме, ядерный взрыв/циунами/прочее

Основные проблемы

- 1) записи, сделанные подтвердившиеся транзакциями не долетели до диска
- 2) запись долетела до диска, транзакция не подтвердилась

Поговорим про первый случай.

Тут нам поможет буфер менеджер. (КАРТИНКА). Транзакции как-то общаются с буфером, буфер в какой-то момент записывает все данные на диск. Запись на диск может произойти значительное позже. Поэтому в момент после транзакции и до записи на диск отключится электричество, то все будет плохо.

В реальности происходит все немного посложнее, но в основе лежат некоторые простые концепции. Как нам восстановить систему после такого сбоя? Нам бы хотелось, чтобы транзакция была бы выполнена. Первое что приходит в голову – просто запомнить транзакцию. Но нам сложно гарантировать детерминированность, в том смысле, что повторный ввод транзакции даст те же результаты.

Журнал(log)

- файл append-only(добавляем только в конец файла данные – с точки зрения ФС)
- место, где тоже появляются изменения
- изменения в журнале записываются до того, как они появляются в таблицах

по сути, СУБД записывает данные дважды

Прежде чем записать X на диск, X появляется в конце журнала в виде некоторой записи.

Таким образом, порядок записи X такой: запись в транзакции, записывает изменения в журнал, потом запись в табличное пространство. Это называется write-ahead logging – опережающее журналирование.

Не является ли это каким-то вредительством?(снижение производительности в 2 раза)

Эти два места независимы, поэтому можно распараллелить. Операции с файлами, открытые на только запись, более эффективны, в то время как табличное пространство – разрозненное. Поэтому, суммарно это все тормозится все на ~ 10%.

Теперь конкретные стратегии, как все записывается в журнал

Undo logging

Записи в журнале.

<START T_i >
<START T_j >
< T_i , X, OLD(X)> OLD(X) – полностью X или дифф. главное, чтобы мы смогли восстановить
<COMMIT T_i >
<ABORT T_j >

Стратегия undo

- запись в журнале предшествует записи в табличном пространстве.
Т.е. t_1 : < T_i , X, OLD(X)>, t_2 : <OUTPUT(X)>, и $t_1 < t_2$ – время
- <COMMIT T_i > появляется после записи всех измененных T_i объектов на диск.

Пример. A = 50, B = 100. Транзакция будет такой:

A *= 2

B *= 2

Что происходит в реальности	Undo log журнал
	<START T>
READ(A)	
A *= 2	
WRITE(A) – запись в буфере	<T, A, 50>
READ(B)	
B *= 2	
WRITE(B)	<T, B, 100>
Flush log?	
OUTPUT(A)	
OUTPUT(B)	
	<COMMIT T>

Восстановление с Undo журналом

- 1) Найти все подтверждившиеся транзакции(есть слово commit)
- 2) сканируя журнал с конца: отменять неподтверждившихся транзакции

Журнал может стать довольно большим. Это и место на диске, и процесс восстановления с большим журналом занимается больше времени.

Checkpointing

контрольные точки

Наивная идея

- 1) подождать пока не будет активных транзакций
- 2) сделать sync
- 3) весь журнал можно выкинуть

Если транзакции немного, то это вполне себе работающая стратегия

<START CHECKPOINT(T_1, \dots, T_n)>
 <START T_{n+1} >
 <COMMIT T_1 >
 <END CHECKPOINT> – когда завершатся транзакции T_1, \dots, T_n .

Если видим с конца и видим окончание чекпоинта? Мы понимаем, что T_1, \dots, T_n закрылись, поэтому нас интересуют другие.

Восстановление с checkpoint

при чтении журнала с конца

- 1) первым нашли <END CHECKPOINT>, значит все интересующие нас транзакции, которые мы бы хотели откатить – начались после <START CHECKPOINT>, поэтому достаточно прочитать до старта чекпоинта.

2) если мы встретили $\langle \text{START CHECKPOINT}(T_1, \dots, T_n) \rangle$, т.е. сбой произошел во время контрольной точки, поэтому читать стоит до начала самой ранней транзакции из $\langle \text{START CHECKPOINT} \rangle$ т.е. из T_1, \dots, T_n .

Redo logging

- 1) В журнал пишет новое значение измененного элемента
- 2) write-ahead logging сохраняется
- 3) никаких OUTPUT пока не появится COMMIT

Рассмотрим на том же примере

$A = 50$, $B = 100$. Транзакция будет такой:

$A *= 2$

$B *= 2$

Что происходит в реальности	Redo log журнал
	$\langle \text{START T} \rangle$
READ(A)	
$A *= 2$	
WRITE(A) – запись в буфере	$\langle T, A, 100 \rangle$
READ(B)	
$B *= 2$	
WRITE(B)	$\langle T, B, 200 \rangle$
	$\langle \text{COMMIT T} \rangle$
Flush log?	
OUTPUT(A)	
OUTPUT(B)	

Это нам гарантирует следующее: если транзакция не подтвердилась, то изменения не запишутся на диск.

С подтвержденными чуть посложнее. Их изменения до диска долетят не сразу.

Восстановление с Redo журналом

- 1) Найти все подтвердившиеся транзакции(есть слово commit)
- 2) сканируя журнал с НАЧАЛА: применять подтвердившиеся транзакции, т.е. записывать новое значение элемента

записываем грязные страницы измененные подтвердившимся транзакциями

При восстановлении

- 1) если есть $\langle \text{END CHECKPOINT} \rangle$ то нас интересует только транзакции из $\langle \text{START CHECKPOINT} \rangle$ и самая ранняя из них
- 2) если завереный checkpoint то нельзя дойти до самой ранней, потому что мы не знаем какие изменения долетели из предыдущих транзакций. Но, если мы дойдем до предыдущего checkpoint – то нужные гарантии мы получим. Т.е. нужно идти до начала самой ранней транзакции из предыдущего checkpoint.

Undo-Redo Logging

- 1) $\langle T, X, \text{Old}(X), \text{new}(X) \rangle$
- 2) write-ahead logging

Применение журнала

Помимо восстановления после сбоя, журнал можно использовать и другим образом.
Если передавать журнал куда-то, то можно получить практически копию БД в реалтайме(репликация)
Еще один плюс Redo: можно получить копию БД на какой-то любой момент времени. Т.е. можно делать копию БД раз в неделю и потом хранить только записи журналов.(непрерывный инкрементальный backup)

Существуют Log based database, который хранят все записи в журнале.