Кластерные и «обычные» индексы MySQL (InnoDB)

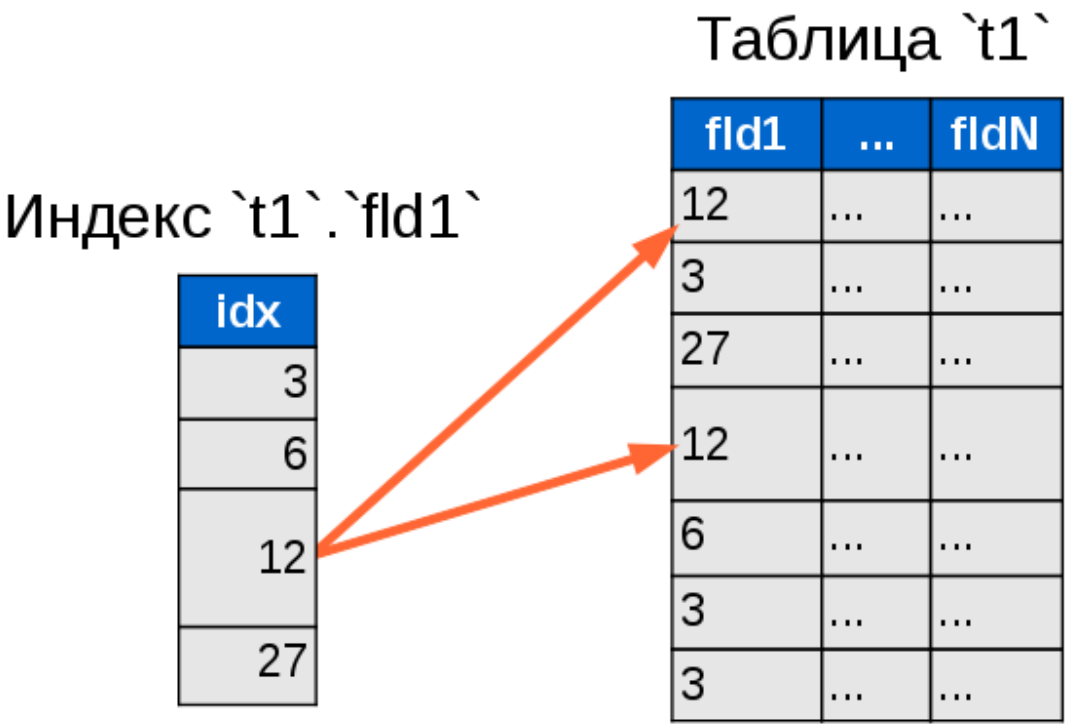
habr.com/ru/post/141767

 freebin 9 апреля 2012 в 23:24

MySQL

Из песочницы

Все мы помним хрестоматийное объяснение «что такое индексы в БД и как они облегчают задачи поиска нужных строк». Уверен, у большинства из вас перед глазами встаёт нечто подобное:



И сразу становится очевидно, насколько меньше данных нужно перелопатить для поиска двух-трёх нужных строк. Гениально. Просто. Понятно.

И лично мне всегда казалось, что улучшать эту схему некуда… Пока я не познакомился с кластерными индексами. Оказалось, что всё не так уж радужно с «обычными» индексами.

Итак, что же такое кластерный индекс, чем он лучше некластерного, и как с ним обстоит дело у MySQL.

Некластерные индексы

Чтобы не запутаться, до поры до времени будем рассматривать простой индекс по одному полю. Упрощённо некластерный индекс можно представить как отдельную таблицу, каждая строка в которой ссылается на одну или несколько строк в таблице с данными. Строки в индексной таблице упорядочены и сгруппированы по значениям ключевых полей. Представим элементарный запрос:

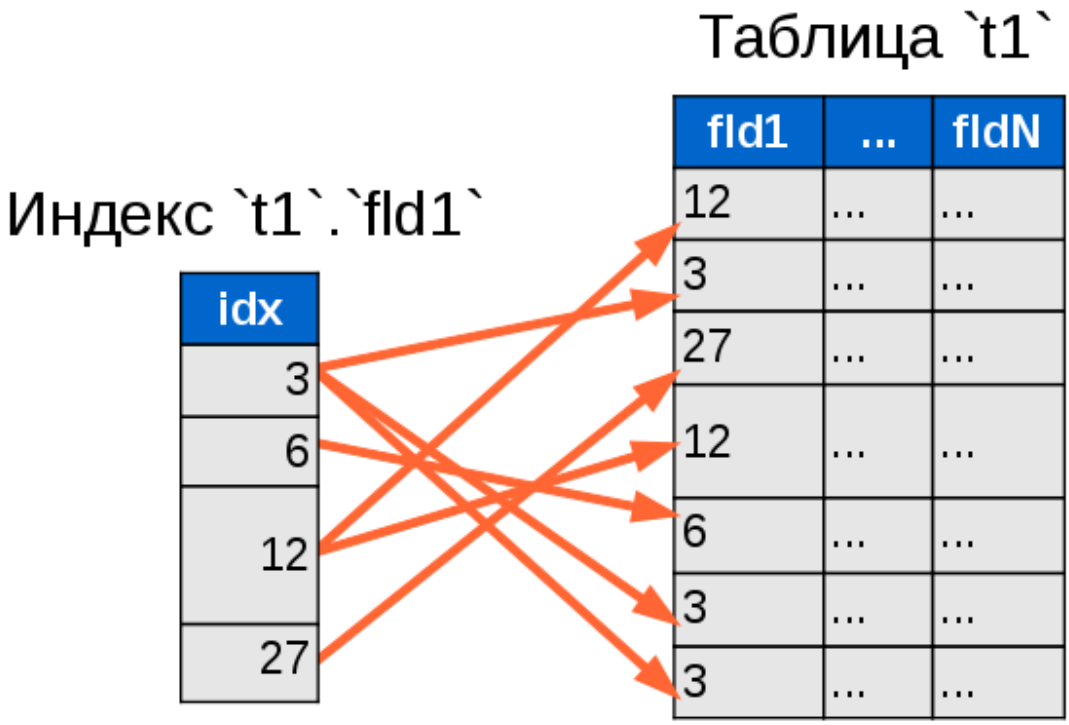
```
SELECT * FROM `t1` WHERE `fld1` = 12;
```

Совсем без индексации будет прочитана и проверена каждая строка, и неудовлетворяющие условию строки просто не попадут в результат. Но прочитаны они будут.

При использовании «обычного», некластерного индекса, задача поиска сильно ускоряется. **Во-первых**, индексная таблица весит много меньше таблицы с данными, а значит элементарно может быть прочитана быстрее. **Во-вторых**, СУБД чаще всего стараются кешировать индексы в оперативную память, которая сама по себе много шустрее жёсткого диска*. **В-третьих**, в индексах отсутствуют дублирующиеся строки. А значит, как только мы нашли первое значение, поиск можно прекращать — оно же и последнее. **В-четвёртых**, данные в индексе отсортированы. А в-третьих и в-четвёртых вместе позволяют использовать алгоритм бинарного поиска (он же метод деления пополам), эффективность которого многократно превосходит простой перебор.

* Если ресурсы позволяют, таблицу данных тоже можно (и нужно) кешировать в оперативную память. Однако индексам и месту для них в оперативной памяти, по понятным причинам, принято уделять больше внимания.

Индексация — великая сила. Но если представить все указатели индексной таблицы на строки в таблице данных ОДНОВРЕМЕННО, получится достаточно сложная «паутина»:



И эта паутина, со множеством пересекающихся стрелок, подводит нас к проблеме (просто таки наглядно её демонстрирует), которую создаёт некластерный индекс.

Фрагментация

Оптимизатор MySQL может принять решение вообще не использовать индексы для поиска по небольшим таблицам (до пары десятков записей — зависит от конкретной структуры данных и индекса). Почему? Потому что поиск простым перебором читает данные последовательно. А указатель в индексе ссылается на разрозненные участки данных. И прыжки по ссылкам из индекса в конечном итоге могут стоить дороже полного перебора.

Итак, что мы имеем на данном этапе эволюции индексирования. Представьте большую, фрагментированную с точки зрения индексации, таблицу. Как данные приходили хаотичными и неотсортированными, так они и сохранялись. Теперь представьте индексную таблицу к ней. И наш старый добрый запрос:

```
SELECT * FROM `t1` WHERE `fld1` = 12;
```

Что происходит? Находится значение в индексе — это быстро и просто — и из таблицы данных читаются строки, на которые этот индекс ссылается. Естественно, при большой фрагментированности таблицы накладные расходы на чтение из разных её частей становятся ощутимыми.

И вот тут-то нам и пригодятся...

Кластерные индексы

Кластерные индексы отличаются от некластерных точно так же, как оглавление книги отличается от алфавитного указателя. Алфавитный указатель (некластерный индекс) для точного слова (значения) даёт точные номера страниц (строки в БД). Оглавление же указывает диапазон страниц, соответствующих определённой главе, в которой уже найдётся искомое слово. Причём каждая глава, если она достаточно велика, может содержать собственное оглавление.

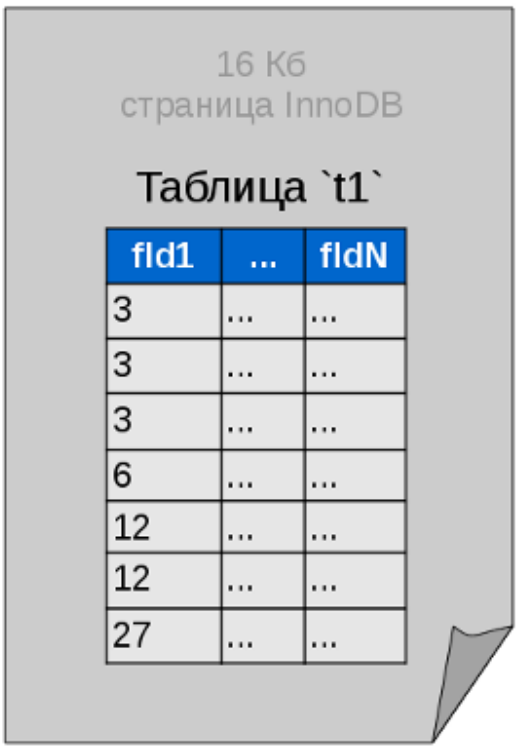
Кластерный индекс — это древовидная структура данных, при которой значения индекса хранятся вместе с данными, им соответствующими. И индексы, и данные при такой организации упорядочены. При добавлении новой строки в таблицу, она дописывается не в конец файла*, не в конец плоского списка, а в нужную ветку древовидной структуры, соответствующую ей по сортировке.

* В разных движках и при разных настройках это может быть вовсе и не конец, и вовсе и не файла. Слово файл здесь означает «некую единицу измерения данных, соответствующую одной таблице», а «конец файла» употребляется как символ последовательной, линейной записи.

Один из самых мощных и производительных движков для MySQL — InnoDB. Тому много причин, и одна из них — кластерные индексы. Проще всего понять как устроены кластерные индексы, если представить их в динамике: как они разрастаются по мере добавления данных, и как начинает ветвиться таблица.

Первый этап: плоский список

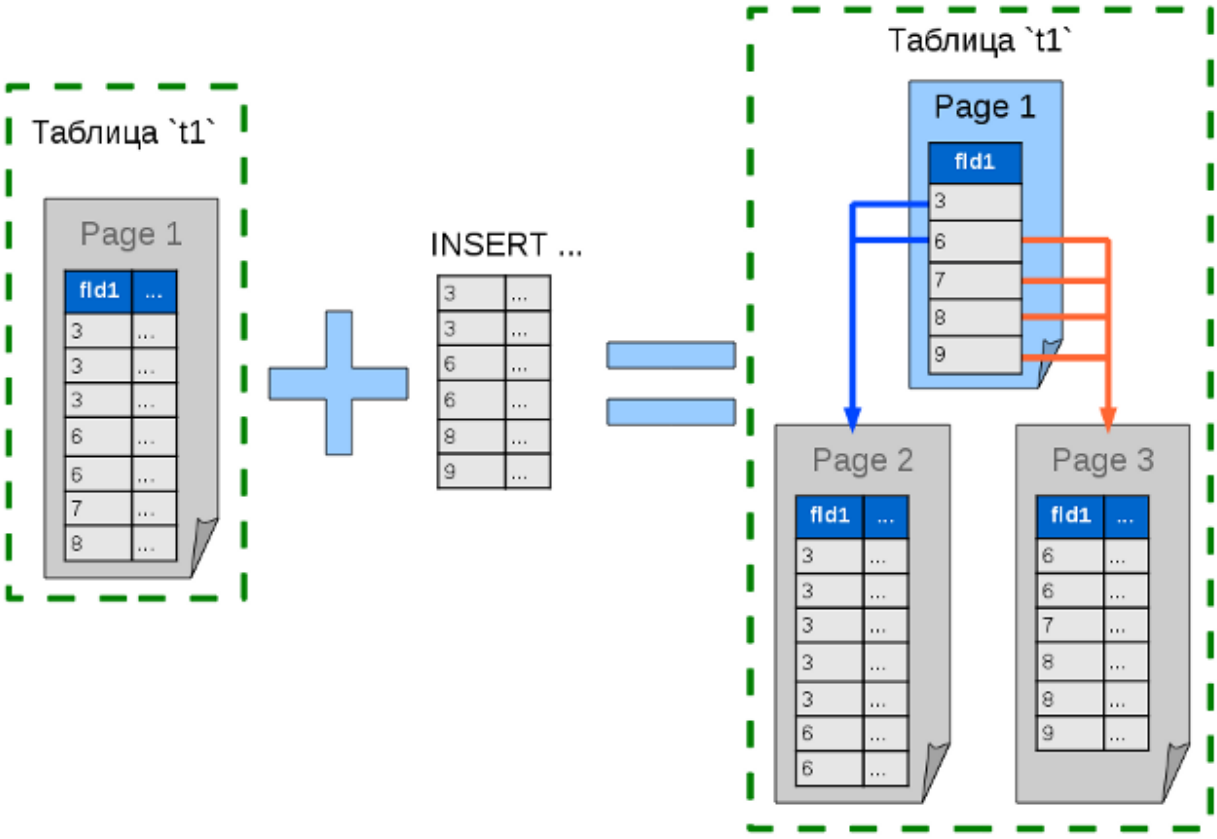
Данные в InnoDB хранятся страницами по 16 Кб. Размер одной страницы — это предельный размер узла нашей древовидной структуры, от которого зависит в какой момент начнётся ветвление. Если вся таблица помещается в одну страницу, то она хранится в виде плоского списка, отсортированного по ключевому полю, без отдельной индексной таблицы.



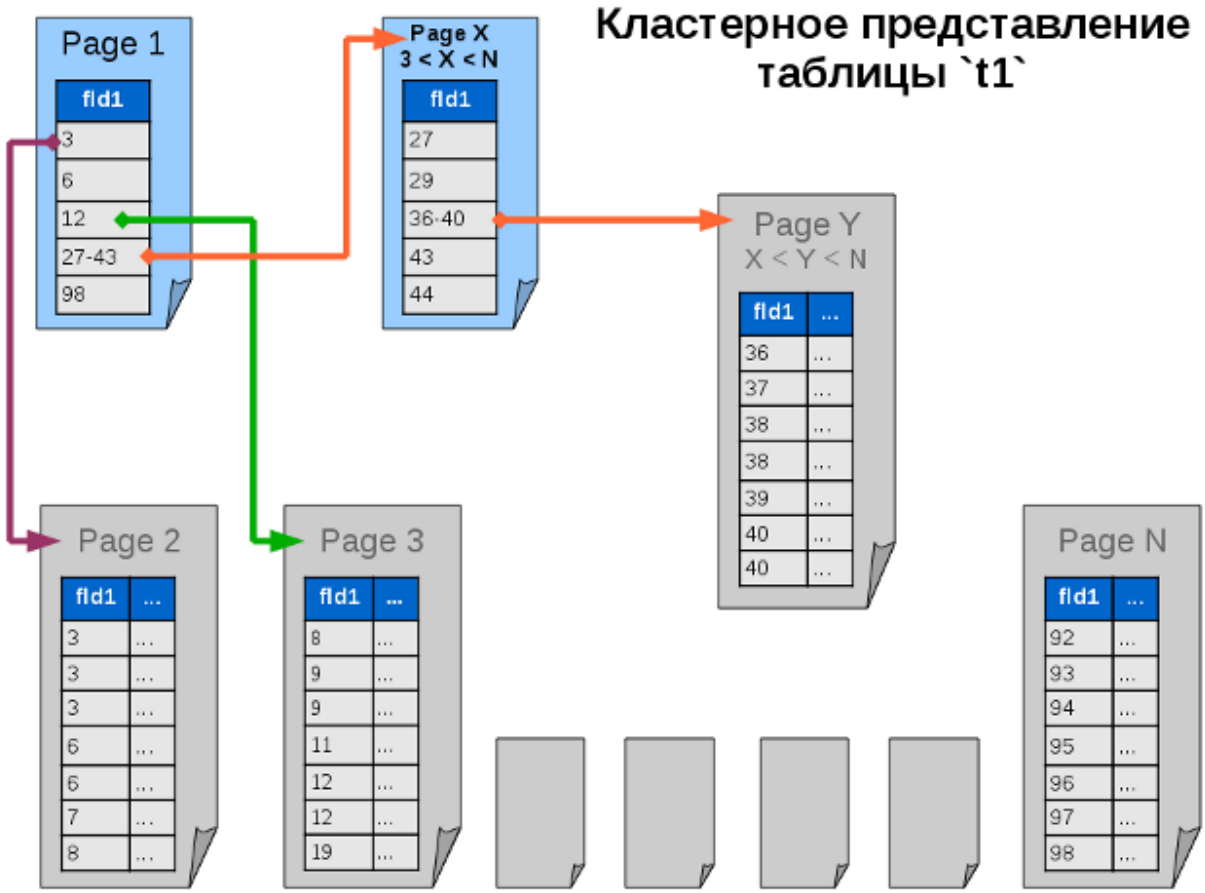
Точно такими же маленькими табличками в будущем будут представлены все наши данные, а соединять их в дерево будут цепочки индексных страниц.

Второй этап: дерево

Когда данные перестают помещаться в одну страницу, список превращается в дерево. Страница с данными разделяется на две, причём в том узле (на той странице), где раньше были данные, теперь располагается индекс, охватывающий обе новые страницы. Конкретный узел такого дерева обязан включать в себя индексы всех дочерних узлов или конечные данные, если узел последний. Узлы могут ссылаться друг на друга только в одном направлении: от родителя к потомку.



По мере добавления всё новых и новых данных, дерево будет усложняться и углубляться. И чем больше оно будет и ветвистее, тем больший выйгрышь даст такая схема хранения данны.



Серые страницы идентичны странице первого этапа — это просто отсортированные данные, листья (конечные узлы) нашего дерева. **Голубые страницы** — это промежуточные узлы дерева, содержащие только индекс и не содержащие данных. **Стрелками** помечены пути поиска определённых значений ключа.

Вспомним наш запрос (зелёная стрелка):

```
SELECT * FROM `t1` WHERE `fld1` = 12;
```

Обращаясь к таблице, запрос попадает на первую страницу и получает индекс, тут же отправляющий его на конечную страницу с данными, где находятся строки, удовлетворяющие критериям поиска. Страница уже прочитана на этапе поиска, все данные собраны, БД может вернуть ответ.

Однако индекс, указывающий на другую страницу, не обязательно ведёт сразу на страницу с данными. Индекс может указывать на страницу с промежуточным индексом. Возможно, при больших объёмах таблицы, БД придётся провести больше итераций поиска, но каждая такая итерация включает минимальный объём данных, а потому в целом всё равно поиск проходит быстрее.

Здесь действует простое правило, актуальное для любого типа индекса: чем разнообразнее данные, тем эффективнее использовать индекс для поиска конкретных значений.

Поскольку данные являются частью индекса, отсортированы и целенаправленно фрагментированы, очевидно что для одной таблицы может использоваться только один кластерный ключ. Из такой, достаточно сложной логики хранения индексов и данных, есть ещё одно важное следствие: операции записи, а особенно изменение имеющихся данных ключевых полей — крайне ресурсоёмкий процесс. Старайтесь использовать для кластерных индексов редко изменяемые поля.

Что касается сложных (составных) кластерных ключей, для них действует абсолютно такая же схема, только сортировка данных осуществляется по двум полям. Сам же индекс мало отличается от некластерного составного ключа.

Кластерные ключи в InnoDB

Здесь всё просто. **Каждая таблица InnoDB имеет кластерный ключ.** Каждая. Без исключения.

Гораздо интереснее, какие поля для этого выбираются.

- Если в таблице задан PRIMARY KEY — это он
- Иначе, если в таблице есть UNIQUE (уникальные) индексы — это первый из них
- Иначе InnoDB самостоятельно создаёт скрытое поле с суррогатным ID размером в 6 байт

До третьего пункта лучше не доводить свой многострадальный сервер, и добавить таки ID самостоятельно.

И не забывайте, что InnoDB во вторичных ключах хранит полный набор значений полей кластерного ключа в качестве ссылки на конечную строку в таблице. Чем больше первичный ключ, тем больше вторичные ключи.

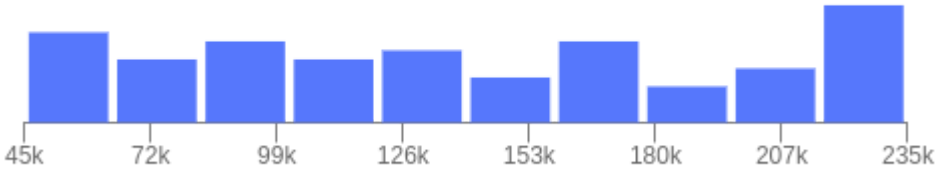
Хабы:

MySQL

Средняя зарплата в IT

120 000 ₽/мес.

Средняя зарплата по всем IT-специализациям на основании 4 821 анкеты, за 1-ое пол. 2021 года Узнать свою зарплату



Комментарии 33

- А каким образом в индексной таблице хранятся ссылки на строки в таблице данных?

Сам первичный (кластерный) ключ хранится вместе с данными и под абстракцию «индексной таблицы» не подходит. А вот вторичные ключи все имеют скрытую часть в виде полного набора полей ключа первичного.

Пример. Есть таблица t1 (InnoDB) с полями k1, k2, i1, i2. Есть первичный ключ по полям k1, k2 (он и будет кластерным). И есть «обычный» индекс (вторичный ключ) по полям i1, i2.

```
CREATE TABLE `t1` (
  `k1` int(10) unsigned NOT NULL DEFAULT '0',
  `k2` int(10) unsigned NOT NULL DEFAULT '0',
  `i1` int(10) unsigned NOT NULL DEFAULT '0',
  `i2` int(10) unsigned NOT NULL DEFAULT '0',
  PRIMARY KEY (`k1`,`k2`),
  KEY `secondary_index` (`i1`,`i2`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

Строки в индексной таблице для вторичного ключа будут представлять собой кортежи вида

```
(`i1`,`i2`, `k1`, `k2`)
```

где первая пара полей (i) используется для поиска второй (k). Когда получена вторая пара (k, образующая первичный ключ), по ней ищутся строки в таблице данных.

Заполним таблицу случайными данными:

```
mysql> SELECT * FROM `t1`;
+-----+-----+-----+-----+
| k1  | k2  | i1  | i2  |
+-----+-----+-----+-----+
| 251 | 762 |  60 |  13 |
| 786 | 490 |  92 | 988 |
| 885 | 385 | 272 | 202 |
| 159 | 403 | 537 | 480 |
| 624 | 341 | 830 | 130 |
| 667 | 372 | 856 | 163 |
+-----+-----+-----+-----+
6 rows in set (0.00 sec)
```

И посмотрим план выполнения простого запроса с фильтрацией:

```
mysql> EXPLAIN SELECT * FROM `t1` WHERE `i1` > 200 AND `i2` < 200\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
         type: range
possible_keys: secondary_index
          key: secondary_index
      key_len: 4
          ref: NULL
         rows: 4
        Extra: Using where; Using index
```

Ключ наш оптимизатором запросов признан годным и в плане выполнения используется. Кроме того, Using index в графе Extra означает, что фактического чтения данных с диска не производилось. Все данные брались прямо из индекса. В графе key указан secondary_index, значит все выбранные данные содержатся в нём.

Для доказательства и интереса ради проведём простой эксперимент. Выполним

```
ALTER TABLE `t1` DROP PRIMARY KEY;
ALTER TABLE `t1` ADD PRIMARY KEY (`k1`);
```

В результате чего поле k2 выпадает из индексов. План выполнения запроса тут же меняется:

```
mysql> EXPLAIN SELECT * FROM `t1` WHERE `i1` > 200 AND `i2` < 200\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
         type: ALL
possible_keys: secondary_index
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 6
        Extra: Using where
```

Обратите внимание на «key: NULL». Поскольку так или иначе придётся читать данные, не входящие в ключ (выбираем-то мы "*"), оптимизатор MySQL принимает решение не использовать ключи вообще, а провести фулскан (type: ALL). Естественно, в Extra больше нет «Using index».

Но стоит нам исключить из выборки поле `k2`, как мы возвращаемся к использованию кластерных ключей:

```
mysql> EXPLAIN SELECT `k1`, `i1`, `i2` FROM `t1` WHERE `i1` > 200 AND
`i2` < 200\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
         type: range
possible_keys: secondary_index
          key: secondary_index
      key_len: 4
          ref: NULL
         rows: 4
        Extra: Using where; Using index
```

На закуску то же самое с MyISAM.

```
CREATE TABLE `t1_isam` (
  `k1` int(10) unsigned NOT NULL DEFAULT '0',
  `k2` int(10) unsigned NOT NULL DEFAULT '0',
  `i1` int(10) unsigned NOT NULL DEFAULT '0',
  `i2` int(10) unsigned NOT NULL DEFAULT '0',
  PRIMARY KEY (`k1`,`k2`),
  KEY `secondary_index` (`i1`,`i2`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1

mysql> SELECT * FROM `t1_isam`;
+-----+-----+-----+-----+
| k1  | k2  | i1  | i2  |
+-----+-----+-----+-----+
| 233 | 203 | 315 | 964 |
| 875 | 485 | 801 | 549 |
| 341 |  58 | 267 | 163 |
|  13 | 574 | 833 | 444 |
| 719 | 262 | 152 | 977 |
| 426 | 201 | 726 |  27 |
+-----+-----+-----+-----+
6 rows in set (0.00 sec)

mysql> EXPLAIN SELECT * FROM `t1_isam` WHERE `i1` > 200 AND `i2` <
200\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1_isam
         type: ALL
possible_keys: secondary_index
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 6
        Extra: Using where
```

Оптимизатор с первой же попытки отказывается использовать ключ, фулскан быстрее. В данном примере MyISAM будет особенно быстр (все поля NOT NULL, все поля фиксированной длины — очень удобно читать).

Ограничиваем выборку полями `i1` и `i2` (они и только они находятся во вторичном ключе в MyISAM).

```
mysql> EXPLAIN SELECT `i1`, `i2` FROM `t1_isam` WHERE `i1` > 200
AND `i2` < 200\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1_isam
         type: range
possible_keys: secondary_index
          key: secondary_index
      key_len: 4
          ref: NULL
         rows: 5
        Extra: Using where; Using index
```

Вуаля!

- Какое-то упрощенное описание. Начнем с того, что индексы могут быть не уникальными. Плоских индексов не бывает.

  Most MySQL indexes (PRIMARY KEY, UNIQUE, INDEX, and FULLTEXT) are stored in B-trees. Exceptions are that indexes on spatial data types use R-trees, and that MEMORY tables also support hash indexes.

  > Описание специально упрощённое. Это достаточно базовые знания о структуре таблиц и индексов, статья писалась для относительных новичков. Мне такой информации в своё время очень не хватало. Именно упрощённой и разжёванной.
  >
  > Что касается «плоской» и не-совсем-такой-как-в-реальности «индексной таблицы» (каковой, строго говоря, вообще не существует) — это абстракция, упрощающая восприятие. Возможно я зря решил так упростить, но лично мне было бы проще воспринимать именно так.
  >
  > Речь здесь именно о InnoDB, а там свои правила: dev.mysql.com/doc/refman/5.0/en/innodb-index-types.html И кластерный индекс создаётся всегда.

- кроме того:

  > в индексах отсутствуют дублирующиеся строки

  в общем виде (так как сформулировано) — неверно. Зависит от реализации. Например, в MyISAM — могут быть.

  > позволяют использовать алгоритм бинарного поиска

  в общем виде (так как сформулировано) — неверно. Зависит от реализации. Например, в MyISAM в индексе по строкам — линейный поиск по странице, а не бинарный.

  И совершенно не затронут BKA — batch key access — который в значительной степени смягчает проблему произвольного доступа к данным.

  Проблема древовидной организации, как в InnoDB, в том, что любой table scan превращается в случайное прыганье по диску. В MyISAM — это последовательное чтение, что намного быстрее. Вообще-то даже в случае с деревом можно бы сделать последовательное чтение, но в InnoDB его не сделали.

  В InnoDB может быть только один кластерный индекс в таблице. А в TokuDB — сколько угодно. Стоило бы упомянуть.
  - Спасибо. Обязательно дополню.

  - > любой table scan превращается в случайное прыганье по диску

    Это еще почему? Страницы же читаются в том порядке, в котором лежат на диске. Страницы промежуточных уровней пропускаются.
    > ну да :)
    > именно это я имел в виду, **можно** сделать чтобы full table scan читал b-tree дерево последовательно. Но в InnoDB это не сделано, там table scan реализован через primary index scan.

- > В-четвёртых, данные в индексе отсортированы.

Я бы упомянул это во первых. Упомянутое же вами во-первых и во-вторых, мне кажется, куда менее

Однако ж и это в-четвертых совсем же не так. Все индексы InnoDB имеют структуру двоичного дерева. И кластерные и не кластерные. А вы эту особенность приписываете только кластерным индексам. Кластерные и обычные индексы отличаются лишь т.н. «полезной нагрузкой». Кластерный индекс в качестве полезной нагрузки несет все поля таблицы, в то время, как обычный лишь значения кластерного индекса. В этом контексте преимущество кластерного индекса лишь в том, что отсутствует дополнительное чтение на вычитку не включенного в некластерный индекс значения.

Упорядоченность же индекса заключается в том, что все листья его двоичного дерева имеют кросс-ссылки на соседние листья. На вашей схеме это важное свойство опущено. Благодаря этой связанности оказывается возможным поиск по диапазонам (range). Т.е. отбор по предикату id > 10 сводится к поиску по дереву значения 10 и дальше, по связям листьев, отбираются все значения, которые больше 10.

Соответственно двоичного поиска, о котором вы упоминаете — здесь не существует. Есть поиск по двоичному дереву, суть совсем другое, хотя, чем то похоже — не спорю.

- В общем — тема интересная, важная. Но, увы, статья получилась не однозначная. Мне затруднительно оценить чего тут больше — введения в заблуждение или же разложения по полочкам.
    - Значит перестарался упрощать материал. Постараюсь исправиться. Спасибо.

- Да, действительно. Если почитать официальную документацию MySQL про индексы, становится понятно, как всё работает.

  После прочтения же данной статьи я запутался. Автор, может быть, ещё не поздно переработать статью? Тем более, что её столько человек добавили в избранное?

  И ещё, ztxn, скажите, а где можно почитать про кросс-ссылки между нодами B-Tree? Документация MySQL только говорит о том, что дерево упорядочено.

> скажите, а где можно почитать про кросс-ссылки между нодами B-Tree?

Я это почерпнул из докуметнации по ораклу. Сейчас, погуглив по ключевому слову «B-tree», я что-то стал сомневаться, особенность ли это самой структуры или же это особенность ее реализации ораклом. Но, с другой стороны, если это не так — как организовывать поиск по диапазонам?

- Первое, что пришло к голову, — просто хранить индекс упорядоченно (на диске, в памяти). Наверное, это очень затратно при изменении индекса, но как минимум это возможно. Наверное, есть и другие способы.

Наверное вы все таки правы.

> If the table has no PRIMARY KEY or suitable UNIQUE index, InnoDB internally generates a hidden clustered index on a synthetic column containing row ID values. The rows are ordered by the ID that InnoDB assigns to the rows in such a table. The row ID is a 6-byte field that increases monotonically as new rows are inserted. Thus, the rows ordered by the row ID are physically in insertion order.

Тут подхрамывает мое знание английского. В последнем слове physically относится к orderd или к insertion order?

> Наверное, это очень затратно при изменении индекса

Я так понял эти издержки пытаются минимизировать оставляя страницы недозаполненными...

Судя по всему вы действительно правы, а я — нет. Спасибо.

- *В последнем слове

  Имелось в виду в последнем предложении.

- > Thus, the rows ordered by the row ID are physically in insertion order.

  Это переводится так: «Таким образом, строки, отсортированные по ID, физически [хранятся] в том порядке, в котором они записывались в таблицу».

  Но заметьте, этот абзац — о дефолтном кластерном индексе, который создаётся, если нет PRIMARY KEY и т.д. Может быть, с обычными кластерными индексами что-то по-другому?

  По крайней мере, я всё ещё не очень представляю себе, как описанное соответствует хранению ***дерева***. Ведь дерево не плоское, и тогда нужно ухитряться хранить подряд, например, сначала ноды верхнего уровня, потом подряд ноды второго уровня... То есть, раскладывать дерево в плоский список при помощи такого себе «поиска в ширину». Второй подход — использовать «поиск в глубину». Однако, я не вижу, как в этом случае можно эффективно искать в диапазоне. В первом подходе вообще не эффективно, во втором иногда прийдётся проходить подряд несколько больших веток (это если хочется использовать то, что записи хранятся подряд).

Я полагаю физически упорядочены листья, не само дерево. Дерево используется для прямого доступа и доступа к первому значению при поиске по диапазону, дальше, при поиске по диапазону — последовательное сканирование листов.

Но мне все равно становится плохо при мысли, что данные физически упорядочиваются по ключу. Что если у нас ключ не монотонно растущий, а, например, — натуральный (номер паспорта, ИНН) или GUID, или составной, связанный с другими наборами данных, как, скажем, при реализации отношения многие-ко-многим… Добавление нового значения может повлечь за собой пересортировку чуть ли не всего набора данных. Если данные действительно физически упорядочены, пожалуй, используя этот движок, следует заведомо отказаться от использования естественных ключей в пользу суррогатных.

- Понятно. То есть, данные хранятся отдельно, дерево поиска — отдельно. Тогда действительно получается, что это B+ tree.

  > Я кажется понял, о чем вы недоумевали в предыдущем посте. B-tree — сбалансированное дерево. Т.е. все его листья расположены на одном уровне глубины. Получается они как бы и в дереве, но как бы и обособленно.
  >
  > На иллюстрации автора, кстати, это не так. Page Y имеет большую глубину нежели Page 2. Но в контексте того, сколько я уже высказал собственных заблуждений в комментариях к этому топику, поостерегусь тут давать оценку еще и тут :D
  >
  > > Судя по вики, разница между B-Tree и B+ tree как раз в том, что в B-Tree каждый нод содержит как данные, так и ссылки на ноды «справа» и «слева» от данных. А в B+ tree только ссылки, а данные как раз хранятся только в листьях.

- Буду смотреть с высот ms sql: у него нет физического порядка в кластерных ключах, в том смысле, что вообще-то порядок есть, но внутри страницы данные хранятся в том порядке, в каком они туда поступили (из ключей данной страницы можно построить такой упорядоченный кусок, что на других страницах не будет данных изнутри этого куска). При добавлении в середину таблицы точно так же произойдёт разделение заполненной страницы пополам и в одну из этих половинок добавятся новые данные и получим в середине данных две наполовину заполненные страницы, остальные страницы никакого действия над собой не увидят (что мало отличается от таблиц без кластерного ключа, разве что разреженность таблиц будет выше). Ещё в пользу отсутствия физического порядка: страница читается полностью (данные и дерево кластерных ключей оказываются в памяти), поиск будет по листьям дерева (иначе зачем нам этот кластерный ключ), из него мы легко найдём нужную строку данных (что в начало страницы запихни его, что в конец — скорость будет одинаковая), так к чему хранить физически упорядоченные строки? Достаточно того, чтоб данные оказались на нужной странице. А если при выборке не используется кластерный ключ, то какая нам разница что данные в страницах упорядочены по нему? Ну и обидно что во всех учебниках делается упор на физический порядок данных.

> При добавлении в середину таблицы точно так же произойдёт разделение заполненной страницы пополам и в одну из этих половинок добавятся новые данные и получим в середине данных две наполовину заполненные страницы, остальные страницы никакого действия над собой не увидят

Я так понимаю, что такое действие возможно, лишь если сосдение листья имеют кроссылки друг на друга. Получается как бы двусвязный список, добавление в середину которого не приводит к переупорядочиванию хвоста списка.

В документации же по mySQL мной лично не найдено ни одной строчки, оставляющей надежду, что дела там обстоят именно так. Более того, там не однократно упоминается физическое упорядочивание. К рассуждениям какие следствия от того могут иметь место быть в результате такой реализации, и как оно вобще может при этом как-то работать, вы и оставлили свой коментарий))

- Только что нашёл в Википедии статью про B+ tree. И там как раз описано, что соседние ноды часто имеют ссылки друг на друга:

  > The leaves (the bottom-most index blocks) of the B+ tree are often linked to one another in a linked list; this makes range queries or an (ordered) iteration through the blocks simpler and more efficient (though the aforementioned upper bound can be achieved even without this addition).

  В документации MySQL'я не расписывается, как именно устроен индекс. Так что возможно, вы изначально были правы. Нужно будет почитать in-depth про устройство индексов MySQL'я.
- ИМХО это особенность реализации ораклом. в myisam точно никаких кросс-ссылок нет, на счет innodb не в курсе.

  в InnoDB как раз есть (насколько я помню)

- В статье автор описывает, что кластерные индексы организованы в виде небинарного дерева. То есть у нода может быть больше двух дочерних нодов. Это не так?

  Спасибо за это замечание. B-tree таки не «двоичное дерево» (binary tree), я использовал не правильный термин. В структуре B-tree нода может иметь более двух дочерних, как и указано в статье.

- НЛО прилетело и опубликовало эту надпись здесь

- Любопытный вопрос возник. А можно как-то сделать частичный индекс таблицы? Есть однородные данные, логично, что должны быть в одной таблице. Индексировать надо по разным полям, так что индексов много и они тяжелые. Но большинство индексов нужны только для последних записей (скажем, по id>1000 или по какому-то другому условию, скажем field IS NULL).

  Как вариант — хранить их в двух таблицах, одна для свежих записей в работе (где нужны все индексы, но индексы короткие и быстрые, потому что таблица короткая), а другая для архива (индекс только по id). Но может быть как-то красивее можно?

  - Можно partitions использовать. Индекс будет создаваться для каждой партиции, конечно, но при выборке будет использоваться только одна:

    mysql> create table t1(f1 int, f2 int, f3 int, updated timestamp, key(f1,f2), key(f2,f3), key(f1,f3))engine=innodb
    -> PARTITION BY RANGE (UNIX_TIMESTAMP(updated)) (
    -> PARTITION p0 VALUES LESS THAN (UNIX_TIMESTAMP('2008-01-01 00:00:00')),
    ...
    -> PARTITION p8 VALUES LESS THAN (UNIX_TIMESTAMP('2010-01-01 00:00:00')),
    -> PARTITION p9 VALUES LESS THAN (MAXVALUE)
    ->);
    Query OK, 0 rows affected (1.15 sec)

    mysql> insert into t1 (f1,f2,f3) values(1,2,3);
    Query OK, 1 row affected (0.08 sec)

    mysql> explain partitions select * from t1 where f1=1 and updated > '2010-01-01 00:00:00'\G
    *************************** 1. row ***************************
    id: 1
    select_type: SIMPLE
    table: t1
    partitions: p9
    type: ref
    possible_keys: f1,f1_2
    key: f1
    key_len: 5
    ref: const
    rows: 152
    Extra: Using where
    1 row in set (0.01 sec)

- В Oracle — да (пользуясь тем что он NULL не хранит в индексе), в MySQL — нет.

  Партиционирование и периодически перемещать партицию в архивную таблицу.

Только полноправные пользователи могут оставлять комментарии. Войдите, пожалуйста.