Базы Данных

Семинар 8

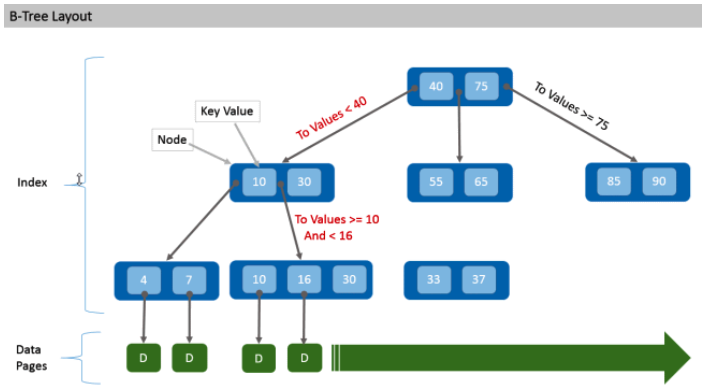
Описание семинара: Оптимизация запросов. Индексы, Партиционирование и сегментирование.

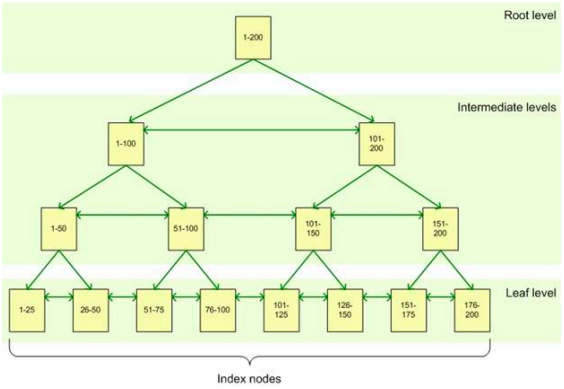
Индексы и проблемы производительности

Индекс – это объект базы данных, обеспечивающий дополнительные способы быстрого поиска и извлечения данных. Индекс может создаваться на одном или нескольких столбцах. Это означает, что индексы бывают простыми и составными. Если в таблице нет индекса, то поиск нужных строк выполняется простым сканированием по всей таблице. При наличии индекса время поиска нужных строк можно существенно уменьшить. К недостаткам индексов следует отнести:

- дополнительное место на диске и в оперативной памяти,
- замедляются операции вставки, обновления и удаления записей.

В SQL Server индексы хранятся в виде сбалансированных деревьев. Представление индекса в виде сбалансированного дерева означает, что стоимость поиска любой строки остается относительно постоянной, независимо от того, где находится эта строка.





Сбалансированное дерево состоит из:

- корневого узла (root node), содержащего одну страницу,
- нескольких промежуточных уровней (intermediate levels), содержащих дополнительные страницы,
- листового уровня (leaf level).

На страницах листового уровня находятся отсортированные элементы, соответствующие индексируемым данным. По мере добавления данных в таблицу индекс будет разрастаться, но по-прежнему оставаться в форме сбалансированного дерева.

Существует два типа индексов:

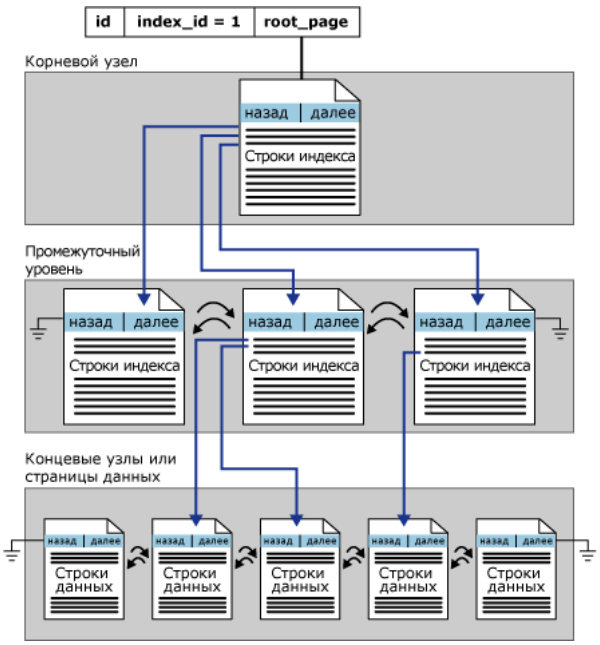
- Кластерные индексы;
- Некластерные индексы, которые включают:
    i. на основе кучи;
    ii. на основе кластерных индексов.

Кластерные индексы

В кластерном индексе таблица представляет собой часть индекса, или индекс представляет собой часть таблицы. Листовой узел кластерного индекса – это страница таблицы с данными. Поскольку сами данные таблицы являются частью индекса, то очевидно, что для таблицы может быть создан только один кластерный индекс. Кластерный индекс является уникальным индексом по определению. Это означает, что все ключи записей должны быть

уникальные. Если существуют записи с одинаковыми значениями, SQL Server делает их уникальными, добавляя номера из внутреннего (невидимого снаружи) 4-х байтного счетчика. Кластерный индекс использовать необязательно. Тем не менее, кластерные индексы часто используют в качестве первичного ключа. На уровне листьев кластерного индекса информация упорядочивается и физически сохраняется на диске в соответствии с заданными критериями сортировки. На каждом уровне индекса страницы организованы в виде двунаправленного связного списка. Вход в корень кластерного индекса дает поле root_page системного представления sys.system_internals_allocation_units.

При добавлении каждой новой записи хранимые данные пересортировываются, чтобы сохранить физическую упорядоченность данных. Если требуется создать новую запись в середине индексной структуры, то происходит обычное разбиение страницы. Половина записей из старой страницы переносится в новую, а новая запись добавляется либо в новую страницу, либо в старую страницу (для сохранения равновесия). Если же новая запись логически попадает в конец индексной структуры, тогда в новую создаваемую страницу помещается только новая строка (информация не делится поровну между двумя рассматриваемыми страницами).



Некластерные индексы на основе кучи

В листьях некластерного индекса на основе кучи хранятся указатели на строки данных. Указатель строится на основе идентификатора файла (ID), номера страницы и номера строки на странице. Весь указатель целиком называется идентификатором строки (RID). С точки зрения физического размещения данных – они хранятся в полном беспорядке. С точки зрения технической реализации нужные записи приходится искать по всему файлу. Вполне может статься, что в этом случае SQL Server придется несколько раз возвращаться к одной и той же странице для чтения различных строк, поскольку определить заранее, откуда придется физически считывать следующую строку нет никакой возможности.

Некластерные индексы, основанные на кластерных таблицах

В листьях некластерного индекса, основанного на кластерных таблицах, хранятся указатели на корневые узлы кластерных индексов. Поиск в таком индексе состоит из двух этапов:

- поиск в некластерном индексе
- поиск в кластерном индексе.

Упрощенный синтаксис явного создания индекса:

CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED] INDEX имя-индекса
ON имя-таблицы-или-представления (список-столбцов)
[INCLUDE (список-столбцов)]
[WITH список-опций]
[ON файловая-группа]

Замечания:

- Для каждого столбца в списке столбцов могут указываться две взаимоисключающие опции ASC/DESC, которые позволяют выбрать способ сортировки индекса. По умолчанию используется вариант ASC, что означает сортировку по возрастанию.
- Предложение INCLUDE указывает неключевые столбцы, добавляемые к ключевым столбцам некластеризованного индекса.
- Следующие за ключевым словом WITH опции являются необязательными. Их можно использовать в любом сочетании. Одни (как например PAD_INDEX, IGNORE_DUP_KEY, DROP_EXISTING, STATISTICS_NORECOMPUTE, SORT_IN_TEMPDB) используются довольно редко, другие (как например FILLFACTOR) существенно влияют на быстродействие и алгоритм работы системы. Описания опций индекса находятся по адресу
- С помощью предложения ON можно задать место хранения индекса. По умолчанию индекс помещается в ту же файловую группу, где находится таблица или представление, для которых он создан.

Пример создания индекса.

```
CREATE UNIQUE NONCLUSTERED INDEX [IX_Address]
ON [Person].[Address]([AddressLine1] ASC,
                      [AddressLine2] ASC,
                      [City] ASC,
                      [StateProvinceID] ASC,
                      [PostalCode] ASC)
WITH (PAD_INDEX = OFF,
      STATISTICS_NORECOMPUTE = OFF,
      SORT_IN_TEMPDB = OFF,
      IGNORE_DUP_KEY = OFF,
      DROP_EXISTING = OFF,
      ONLINE = OFF,
      ALLOW_ROW_LOCKS = ON,
      ALLOW_PAGE_LOCKS = ON)
ON [PRIMARY]
```

Индексы, создаваемые вместе с ограничениям

Такой тип индексов часто называют «связанными индексами». Связанные индексы создаются при добавлении одного из следующих двух типов ограничений:

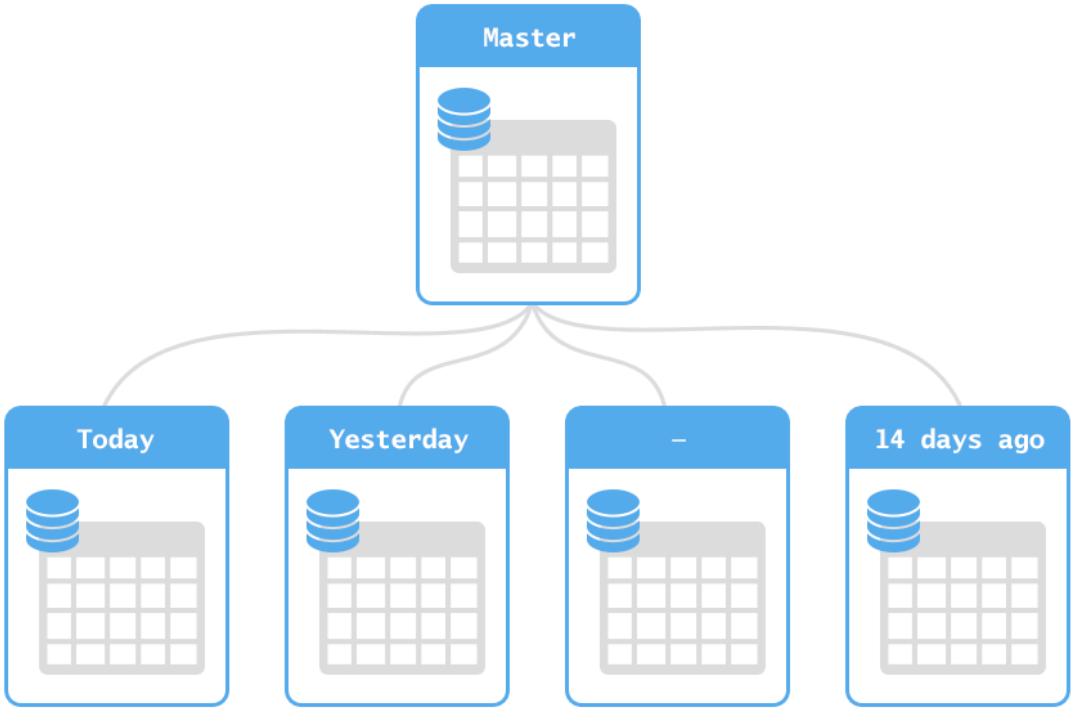
- ограничения первичного ключа (PRIMARY KEY);
- ограничения уникальности (UNIQUE).

Пример создания индекса.

```
CREATE TABLE [Person].[Address](
    [AddressID] int IDENTITY(1,1) NOT NULL,
    ...
    CONSTRAINT [PK_Address_AddressID] PRIMARY KEY
        CLUSTERED([AddressID] ASC)
    WITH (PAD_INDEX = OFF,
          STATISTICS_NORECOMPUTE = OFF,
          IGNORE_DUP_KEY = OFF,
          ALLOW_ROW_LOCKS = ON,
          ALLOW_PAGE_LOCKS = ON)ON [PRIMARY]
) ON [PRIMARY]
```

Партиционирование

Партиционирование – это метод разделения больших (исходя из количества записей, а не столбцов) таблиц на много маленьких. И желательно, чтобы это происходило прозрачным для приложения способом.
Одной из редко используемых фич PostgreSQL является тот факт, что это объектно-реляционная база данных. И «объект» здесь ключевое слово, потому что объекты (или, скорее, классы) знают то, что называется «наследование». Именно это используется для партиционирования.



Давайте попробуем произвести настоящее партиционирование. Для начала нам нужно решить, каким будет ключ партиционирования – другими словами, по какому алгоритму будут выбираться партиции.
Есть пара наиболее очевидных:

- партиционирование по дате – например, выбирать партиции, основываясь на годе, в котором пользователь был создан
    - достоинства:
        - легко понять;
        - количество строк в данной таблице будет достаточно стабильным;
    - недостатки:
        - требует поддержки – время от времени нам придётся добавлять новые партиции;
        - поиск по имени пользователя или id потребует сканирования всех партиций;
- партиционирование по диапазону идентификаторов – например, первый миллион пользователей, второй миллион пользователей, и так далее
    - достоинства:

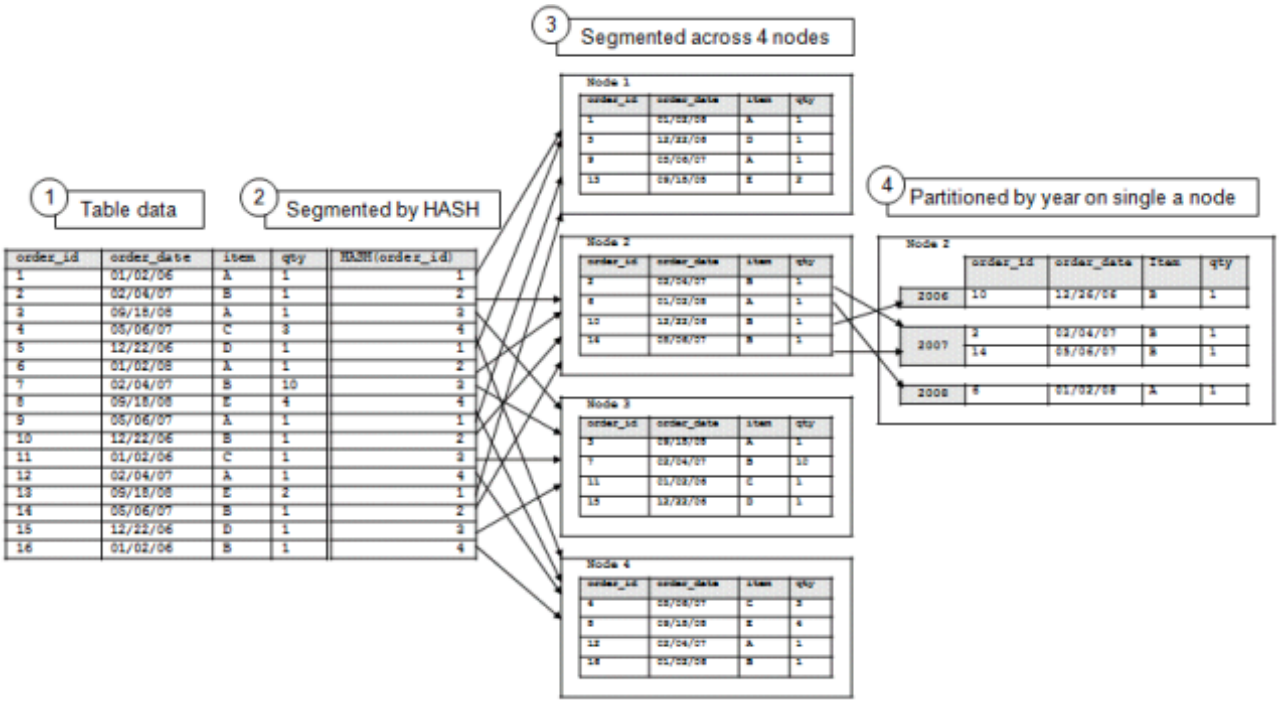
        - легко понять;
        - количество строк в партиции будет на 100% стабильным;
    - недостатки:
        - требует поддержки – время от времени нам придётся добавлять новые партиции;
        - поиск по имени пользователя или id потребует сканирования всех партиций;
- партиционирование по чему-нибудь другому – например, по первой букве имени пользователя.
    - достоинства:
        - легко понять;
        - никакой поддержки – есть строго определенный набор партиций и нам никогда не придется добавлять новые;
    - недостатки:
        - количество строк в партициях будет стабильно расти;
        - в некоторых партициях будет существенно больше строк, чем в других (больше людей с никами, начинающимися на “t*", чем на “y*»);
        - поиск по id потребует сканирования всех партиций;

Есть еще пара других, не так часто используемых вариантов, вроде «партиционирования по хэшу от имени пользователя».

```
CREATE TABLE measurement (
    city_id         int not null,
    logdate         date not null,
    peaktemp        int,
    unitsales       int
) PARTITION BY RANGE (logdate);
```

Сегментирование

Для распределения данных по кластерам используется их сегментация(Segmentation), а точнее сегментация проекций(Projection) в которых они находятся. Задача разработчика — подобрать такой список полей и/или такую функцию(например, хэш-функцию), благодаря которым данные равномерно распределятся по нодам кластера. HP Vertica рекомендует использовать встроенные функции HASH и MODULARHASH для этих целей.
Для обеспечения K-Safety > 0 создаются сообщные проекции(Buddy Projection). Проекции называются сообщными, если они имеют в себе одинаковые наборы полей и одинаковое выражение сегментации, но хранятся на разных нодах. Сообщные проекции позволяют создать те самые реплики, которые позволяют работать БД в выбранном режиме K-Safety.



Рассмотрим K-Safety, сегментацию, и сообщные проекции на практическом примере.
Дано: Кластер на 3-х нодах. K-Safety = 1.

Создадим простую таблицу:

```
CREATE TABLE test (
    n integer primary key
);
```

Вообще, синтаксис создания таблицы позволяет включать в себя выражения влияющие на проекцию(order, hash, ksafe…), для того чтобы проекции создавались автоматически в момент создания таблицы, но в этом примере я создаю проекции вручную.

Далее создадим две сообщные проекции.

```
CREATE PROJECTION p_test_b0 AS
SELECT n FROM test_order BY n
SEGMENTED BY hash(n) ALL NODES;
```

```
CREATE PROJECTION p_test_b1 AS
SELECT n FROM test_order BY n
SEGMENTED BY hash(n) ALL NODES offset 1;
```

Во-первых, выбрано сегментирование по функции hash от поля n. Во-вторых, те же данные, которые хранятся в проекции p_test_b0 на текущем ноде, хранятся в проекции p_test_b1(создана с offset 1) на следующем ноде. Иными словами текущий нод содержит в сегменте проекции p_test_b1 данные предыдущего нода.