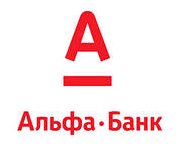
**Обучение оптимизации для разработчиков**



# Введение

Данное обучение предназначено для разработчиков и сотрудников поддержки, желающих более глубоко познать возможности оптимизации в Oracle. Перед прохождением данного обучения необходимо полностью освоить предыдущий материал – обучение оптимизации запросов, предназначенный для аналитиков, так как повторно изложенный материал уже не будет рассматриваться.

# Индексы

Индексы – тот инструмент, который при правильном применении позволяет ускорять запросы (также update и другие операции) в десятки раз, но также при неуместном использовании могут привести к сильной деградации производительности. Поэтому необходимо знать:

* основные виды индексов и понимать их отличия
* методы чтения индексов
* особенности работы с индексами

Первое, что необходимо знать при работе с индексами – в партиционированных таблицах индекс может быть:

* локальным (то есть также делится на секции как и таблица)
* глобальным.

Основное отличие в том, что при обновлении данных в одной из партиций таблицы в локальном индексе обновляются данные только одной партиции. В то время как в глобальном индексе происходит полная перестройка индекса.

**Поэтому в партиционированных таблицах индекс всегда необходимо создавать локальным!**

**Индекс в виде B-дерева** - обычный индекс, является наиболее распространенным. Предназначен для частых выборок по атрибуту (или нескольким атрибутам) с большим количеством уникальных значений. При этом необходимо знать:

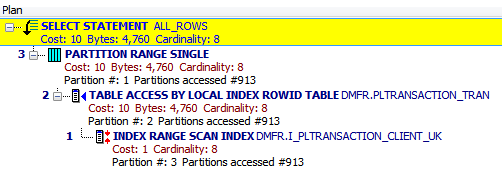
* необходимо быть аккуратным, если есть значения, селективность которых сильно отличается от других. Например, если у нас 1000 уникальных значений, а одно из них занимает более 10%, то его чтение по индексу становится невыгодным
* если индекс составной, то, как правило, первым указывается наиболее селективный (больше всего уникальных значений)
* индекс снижает производительность обновления данных, иногда катастрофичкски!
* невозможно одновременно использовать в запросе более одного индекса на таблицу
* в уникальном индексе данные хранятся в отсортированном виде

Необходимо различать следующие виды индексного чтения:

1. По диапазону (обычное)

*Пример:*

select \* from dmfr.pltransaction\_tran where value\_day = '01.07.2013' and client\_uk = 8386842762

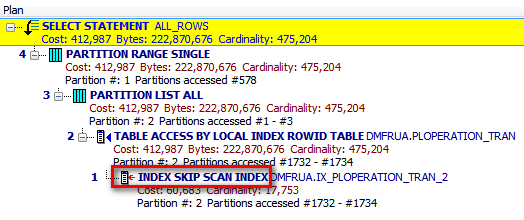


1. Происходит сканирование индекса I\_PLTRANSACTION\_CLIENT\_UK (913 партиции) по диапазону в поисках всех подходящих значений
2. По полученным rowid (о котором речь будет чуть позже) происходит чтение в DMFR.PLTRANSACTION\_TRAN и выборка подходящего значения

2. Skip-scan (если в фильтрации не участвует первый атрибут данного индекса)

*Пример:*  
select \* from dmfrua.ploperation\_tran where value\_day = '31.07.2013'

and mprgroup\_ncode = 4 and budgetitem\_ncode = 10;



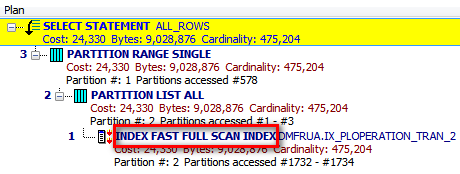
Индекс IX\_PLOPERATION\_TRAN\_2 состоит из следующих атрибутов - PROFITCENTER\_MARK\_CCODE, BUDGETITEM\_NCODE, MPRGROUP\_NCODE. Видим, что в запросе в фильтрации не участвует первый атрибут индекса - PROFITCENTER\_MARK\_CCODE, поэтому получено сканирование такого рода.

К таким сканированиям необходимо относиться осторожно, так как они работают заведомо менее эффективно.

1. Index Fast Full Scan (быстрое полное сканирование индекса)

Возможно, если все запрашиваемые атрибуты таблицы (в том числе участвующие в фильтрации) можно прочитать в одном индексе.

Пример:  
select PROFITCENTER\_MARK\_CCODE from dmfrua.ploperation\_tran p where value\_day = '31.07.2013' and mprgroup\_ncode = 4 and budgetitem\_ncode = 10;



Чтение аналогично полному сканированию таблицы, но здесь происходит полное сканирование индекса (который весит гораздо меньше чем целая таблица), то есть отсутствует чтение данных из таблицы по rowid. Опасаться такого индексного чтения не стоит.

**Битмаповый индекс** – является альтернативой индексу в виде В-дерева. Предназначен для атрибутов с малым количеством уникальных значений (например, пол – М или Ж).

Чем меньше уникальных значений, тем быстрее он работает!

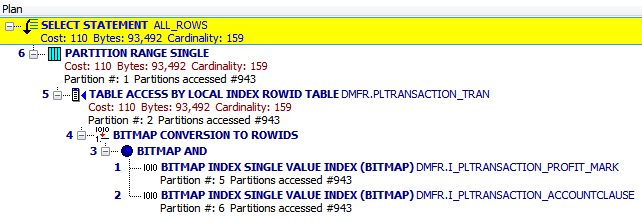
Имеет следующий недостаток – если одна сессия начинает вставку данных в таблицу, то она блокирует всю таблицу и не дает другим сессиям закончить свою вставку. Что зачастую делает невозможным его использование в OLTP-системах.

Его важной особенность является то, что на таблицу возможно наложение одновременно нескольких битмаповых индексов:

*Пример:*

select \* from dmfr.pltransaction\_tran p where value\_day = '31.07.2013'

and accountclause\_mark\_uk = 5216025329 and profitcenter\_mark\_uk = 5210762274;



Видим, что индексы битмаповые и одновременно на таблицу накладываются два индекса – 1) по profitcenter\_mark\_uk и 2) по accountclause\_mark\_uk

Это часто приводит к существенной оптимизации запросов. Но для этого необходимо чтобы этот параметр был включен на базе:

select value from v$parameter where name = 'star\_transformation\_enabled';

Либо установить его на уровне сессии:

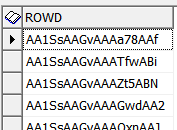
alter session set star\_transformation\_enabled = true;

**ROWID:**

Уникальный идентификатор строки. Показывает физический адрес строки, то есть в каком блоке данных лежит строка и ее идентификатор в блоке. Если обратиться к таблице по rowid, то будет прочитан всего один блок данных, что очень быстро. При этом уже необязательно вносить фильтр по партиционированию.

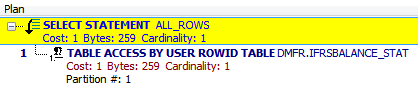
*Пример:*

select rowid as rowd from dmfr.ifrsbalance\_stat where value\_day = '01.01.2013';



Так мы получаем идентификаторы строк (псевдоним обязателен).

Выполнив select \* from dmfr.ifrsbalance\_stat where rowid = 'AA1SsAAGvAAAa78AAf';



Получаем результат запроса за доли секунды, хотя таблица остатков весит более терабайта.

**Массовый DML при наличии индекса.**

Обязательно необходимо знать, что индексы снижают скорость операций массового DML. Причем увеличение непропорционально количеству индексов. Поэтому необходимо всегда придерживаться следующих правил:

1. При инициализирующей загрузке (также обычно при больших update-ах) индексов на таблице быть не должно, их необходимо создавать после DML
2. В частном случае если загружаем одну или несколько партиций в фактовой таблице и в ней есть битмаповые или неуникальные индексы, то по загружаемым партициям индексы необходимо отключать и после загрузки ребилдить отключенные партиции. Выполняется командой alter index unusable или alter index unusable partition

# Распространенные хинты

1. APPEND

Прямая вставка в таблицу. Позволяет увеличить скорость вставки данных в таблицу за счет избавления от логирования вставки и проверки свободного места в блоках.

*Пример:*

insert */\*+ append\*/*

into table1 select \* from table2

**Важно!** Вставка всегда идет поверх уровня high watermark (уровень до какого записаны данные в таблицу), при этом оператор delete не меняет этот уровень, поэтому сочетание delete + insert /\*+ append\*/ приведет к разрастанию таблицы, поэтому они несовместимы.

1. Leading

Хинт, с помощью которого можно исправлять порядок чтения и соединений.

*Пример:*

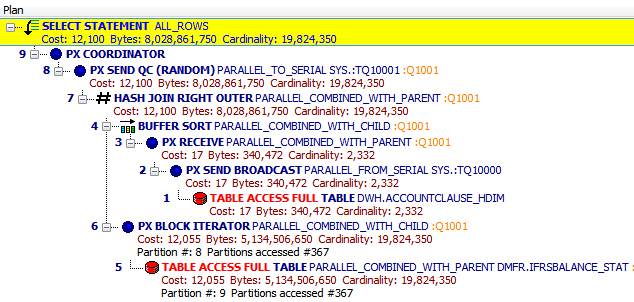
select \* from

dmfr.ifrsbalance\_stat bal

left join dmfr.accountclause\_sdim ac on

ac.uk = bal.accountclause\_bal\_uk

where value\_day = '01.01.2013'



Так выглядит план запроса без хинта. Если добавить хинт, то он преобразуется так:

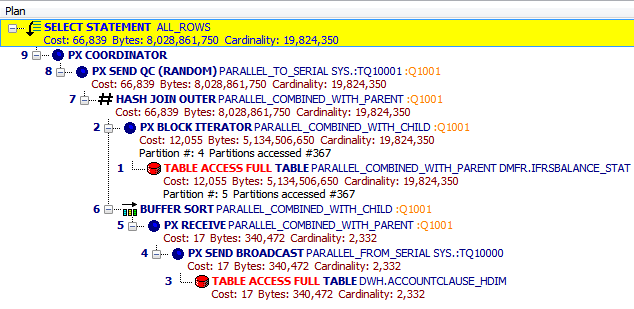
select */\*+ leading(bal)\*/*\* from

dmfr.ifrsbalance\_stat bal

left join dmfr.accountclause\_sdim ac on

ac.uk = bal.accountclause\_bal\_uk

where value\_day = '01.01.2013'



Видим, что порядок соединения поменялся.

1. Ordered

Является аналогом хинта leading, но при этом является более жестким. Поэтому лучше пользоваться гибким хинтом leading.

1. First\_rows

Подсказывает оптимизатору использовать режим оптимизации для более быстрого получения первых строк. По умолчанию на базе используется метод оптимизации ALL\_ROWS, то есть вариант оптимизации для более быстрого получения полного набора строк.

1. Dynamic\_sampling

Можно использовать для таблиц, для которых, по вашему мнению, оптимизатор ошибочно определяет избирательность по условиям запроса, например, в случае отсутствии гистограмм для столбцов с неравномерно распределёнными значениями. Также можно использовать в случае если есть вероятность что статистика на момент запуска запроса не собрана или устаревшая.

Рекомендуется использовать при чтении диапазона большой фактовой таблицы. И нет смысла в использовании при чтении одной партиции, так как данный хинт работать не будет.

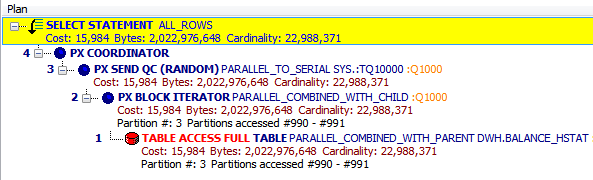
В скобках указывается, для какой таблицы производить сбор статистики (если не указано, то будет производиться для всех таблиц запроса) и уровень сбора статистики от 0 (не собирать статистику) до 10 (собирать статистику по всей таблице). Стоит отметить, что на уровне 10 на большой таблице сбор статистики будет проходить долго, поэтому этот уровень стоит использовать редко.

*Пример:*

select \* from

dwh.balance\_hstat b

where value\_day between '04.09.2013' and '05.09.2013'

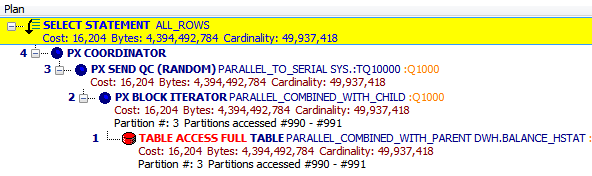


Видим ошибочную кардинальность 23 млн.

select */\*+ dynamic\_sampling(b 4)\*/*\* from

dwh.balance\_hstat b

where value\_day between '04.09.2013' and '05.09.2013'



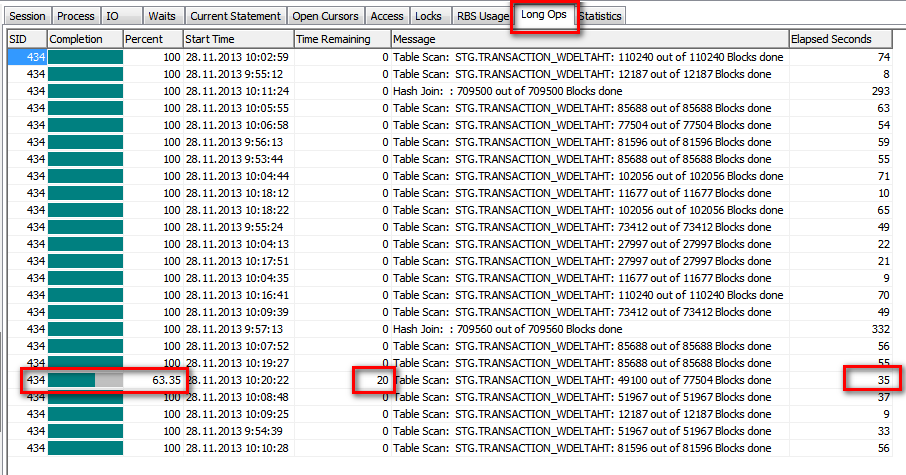
Видим более правильную кардинальность 50 млн.

# Что в данный момент выполняется в запросе?

Зайдя в session browser, найдя свой запрос и перейдя на вкладку Long Ops можно посмотреть, что в данный момент выполняется в запросе, процент выполнения данной операции, сколько времени уже выполняется данная операции и примерное время, которое еще потребуется на данную операцию и некоторые прошедшие операции (они со временем могут удаляться). Сразу стоит отметить, что мы не сможем увидеть если:

1. Выполняются быстрые действия, которые не успевают подхватываться
2. Ничего не выполняется в данный момент по какой-то причине

Пример:



Видим, что у данного запроса в данный момент идет сканирование STG.TRANSACTION\_WDELTAHT (партиций), данная партиция уже прочитана на 63.35 процента, время чтения этой партиции уже составляет 35 секунд и примерное оставшееся время 20 секунд.

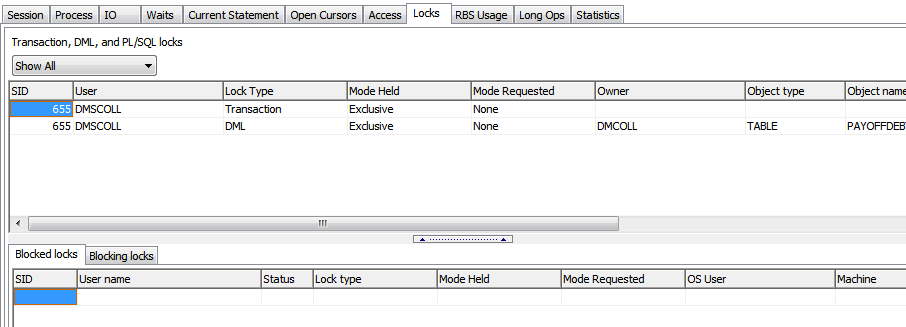
Стоит отметить, что Time Remaining иногда отображает некорректное время, происходит в случае если оптимизатор где-то ошибся, к примеру из-за неактуальной статистики думает что прочитать нужно один объем данных, а приходится совершенно другой.

Кроме чтения таблиц здесь можно увидеть происходящие сортировки или соединения хэшом. Они имеют соответствующие записи в колонке Message.

# Блокировки

Самый удобный способ посмотреть, какие блокировки накладываются данным скриптом или кто блокирует выполнение данного скрипта можно на вкладке Locks.

Пример:



В верхней половине отображаются блокировки, накладываемые данным запросом. Видим, что в данном примере наш запрос наложил эксклюзивную блокировку на таблицу.

В нижней половине отображаются запросы, которые блокируют данный скрипт. Поскольку в нашем примере пусто, то наш запрос никто не блокирует. Иначе можно было увидеть SID блокирующего запроса.

# Основные ожидания

На вкладке Waits можно посмотреть, на чем в данный момент висит выполнение нашего запроса – прямое чтение, чтение индекса, ожидание темпового пространства и тд.

Основные ожидания, на которые стоит обращать внимание:

* direct path read temp – использование дискового пространства temp по причине нехватки выделенной области PGA. Приводит к ощутимой просадке производительности.
* Db file sequential read – чтение по индексу или одноблочное чтение (например по ROWID)
* Db file scattered read – полное сканирование какого-либо объекта
* Direct path read – аналогично предыдущему, но чтение идет в обход буферного кэша
* Free buffer waits – ожидание буферного кэша, который в данный момент не справляется с нагрузкой (большие объемы чтений или DML, идущих через буферный кэш)

Если одно из ожиданий явно преобладает над другими, значит необходимо разбираться в его причине. К примеру, большой db file sequential read скорее всего означает, что где-то идет ошибочное чтение по индексу, например, из-за сбившей кардинальности.

# TEMP, PGA, SGA

Иногда необходимо посмотреть, сколько потребляет тот или иной запрос пространства TEMP, потребляет ли вообще. Если потребляет слишком много, то запрос явно нуждается в оптимизации.

Посмотреть можно с помощью следующего скрипта (им же видим, сколько запросу выделено PGA):

select s.sid,

s.username,

s.sql\_id,

segtype,

round((blocks\*16384)/1024/1024) temp\_used,

round(PGA\_ALLOC\_MEM / 1024 / 1024, 2) pga\_alloc\_mem,

round(PGA\_USED\_MEM / 1024 / 1024, 2) pga\_used\_mem,

round(PGA\_FREEABLE\_MEM / 1024 / 1024, 2) pga\_freeable\_mem

from v$sort\_usage u, v$session s, v$process p

where u.session\_addr (+) = s.saddr

and s.paddr= p.ADDR;

Сделав соответствующие сортировки можно также увидеть запрос, потребляющий много PGA или «забивающего» TEMP. Если запрос потребляет много TEMP’а, то он гарантированно будет долго работать, создавая высокую нагрузку и его необходимо оптимизировать.

Стоит напомнить, что:

1. TEMP – табличное пространстве на диске, которое используется в случае если для операций сортировки или хэш-джойна не хватило выделенного запросу PGA.
2. PGA (Program Global Area) – область в оперативной памяти, выделяемая запросам для выполнения сортировок и хэш-соединений. Общий размер на базе можно посмотреть запросом:

select display\_value from v$parameter where name = 'pga\_aggregate\_target'

Слишком маленький размер приведет к постоянному потреблению запросами TEMP

1. SGA (System Global Area) – область в оперативной памяти, в которой содержатся:
2. буферный кэш (хранение часто используемых данных для их быстрого чтения, размер можно посмотреть select display\_value from v$parameter where name = 'db\_cache\_size'). Чем больше размер, тем больше данных может там поместиться, но при этом постепенно снижается скорость доступа к данным из кэша, поэтому нужен определенный баланс.
3. shared pool (хранение планов запросов для снижения количества жестких разборов запросов, размер select display\_value from v$parameter where name = 'shared\_pool\_size')

# BIND – переменные (связываемые переменные)

Каждый раз при построении плана запроса оптимизатор оценивает все возможные варианты выполнения запроса и выбирает оптимальные на его взгляд. Полученный план на какое-то время (зависит от того как работает система) сохраняется в области под названием shared pool. Если при выполнении запроса есть сохраненный план в shared pool, то он будет взят оттуда, и оптимизатор не будет снова оценивать все возможные планы выполнения.

Но нужно понимать, что для оптимизатора запросы вида:

1. Select \* from table1 where column\_name1 = a;
2. Select \* from table1 where column\_name1 = b;

являются разными, и для второго будет происходить построение плана запроса заново и оба плана будут храниться в shared pool.

Это абсолютно нормально для редко выполняющихся запросов. Более того, для сложных и редко запускаемых запросов это хорошо, так как изменение фильтрации в одном месте может потребовать полного изменения плана. Но если мы запускаем их достаточно часто и хотим чтобы каждый выполнялся очень быстро (например, в OLTP системе или мы запустили построчный цикл), при этом знаем, что каждый из них должен иметь одинаковый план выполнения запроса, то такой вариант становится неприемлимым, так как жесткий разбор запроса (то есть построение для него отдельного плана) является трудоемкой операцией. Это увеличивает время выполнения нашего кода и значительно повышает нагрузку на CPU.

Решением такой проблемы является использование bind-переменных. В этом случае жесткий разбор запроса происходит только для первого выполнения запроса. Все остальные выполнения используют имеющийся в shared\_pool план.

Синтаксис такой:

1. Select \* from table1 where column\_name1 = :x using a;
2. Select \* from table1 where column\_name1 = :x using b;

# Компрессия

Компрессия – это возможность сократить объем, занимаемый таблицей. Обычно показатель составляет 1,5 – 2 раза, иногда более. Что соответственно может снизить время, необходимое на чтение таблицы.

При этом нужно понимать, что оператор UPDATE приводит к раскомпрессии данных. Поэтому компрессия обычно не применяется для таблиц с часто изменяющимися данными.

На чтение скомпрессированной таблицы уходит меньше времени, так как нужно прочитать меньший объем. Но зависимость не прямая, так как несколько повышается нагрузка на CPU из-за необходимости дополнительной обработки данных.

**Важно!** Загрузить скомпрессированные данные можно только с помощью подсказки /\*+ append\*/, без которой данные не будут ложиться в сжатом виде.

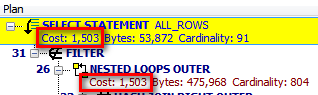
Можно таблицу (партицию) в случае необходимо сжать повторно командой alter table move. Если она еще нескомпрессирована, то alter table move compress.

# Дополнение о планах запросов

В прошлом обучающем курсе было дано достаточно много информации о планах запросов. Но необходимо добавить некоторую информацию, знание которой бывает зачастую необходимо.

1. Стоимость запроса – оценка оптимизатором сколько ресурсов и времени ему придется потратить на выполнение данного запроса (нагрузка на CPU, диски, время и тд). Основывается на статистике как используемых объектов (таблицы, индексы и пр.), так и на статистике базы (текущая загруженность, показатели времени отклика системы и пр.).

Обозначается в плане как cost (как общая так и для каждого шага):



Оптимизатор всегда выбирает план с минимальной стоимостью по его оценке. Но не всегда оптимально, бывает, что изменение плана с помощью хинтов приводит к увеличению оценочной стоимости, но план становится более правильным. Поэтому на стоимость внимание обращать стоит, но не как на догму.

1. Иногда возникают вопросы плана «не было статистики на объекте, что приводило к плохому плану, статистику собрали, а план не изменился, почему?». Ответ следующий – план запроса сохранен в shared pool и оптимизатор взял имеющийся вместо того, чтобы заново производить перестроение плана. Необходимо выполнить следующую команду:

alter system flush shared\_pool

# Распространенные примеры оптимизации

В этом и предыдущем обучающем материале было приведено много примеров распространенных способов оптимизации запросов. Главное это понимать где узкое место (при сомнении проверяются все потенциально узкие места). Рассмотрим еще два примера. Первый – распространенный случай, требующий оптимизации, а второй – пример пошагового чтения сложного плана с дальнейшей оптимизацией.

1. Соединение с небольшим справочником без жесткого условия соединения.

По возможности стоит унести это соединение в подзапрос с другим маленьким справочником, с которым есть хорошее условие джойна.

*Пример:*

select pl.\*, adj.ncode, t.ncode as ncode\_t

from dmfr.pltransaction\_tran pl

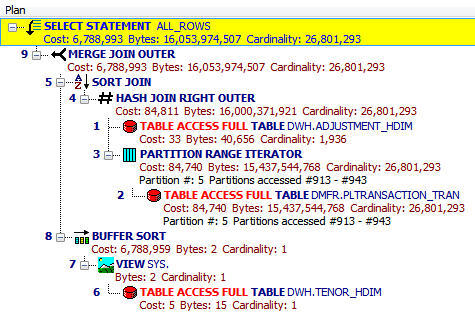
left join dmfr.adjustment\_sdim adj on

pl.adjustment\_mark\_uk = adj.uk

left join dmfr.tenor\_ldim t on

t.ccode = 'Y25'

where pl.value\_day between '01.07.2013' and '31.07.2013'



Получили сортировку 16 гигабайт, такой запрос будет выполняться достаточно долго.

Переписываем так:

select pl.\*, adj.ncode, adj.ncode\_t

from dmfr.pltransaction\_tran pl

left join (select adj.ncode, adj.uk, t.ncode as ncode\_t

from dmfr.adjustment\_sdim adj

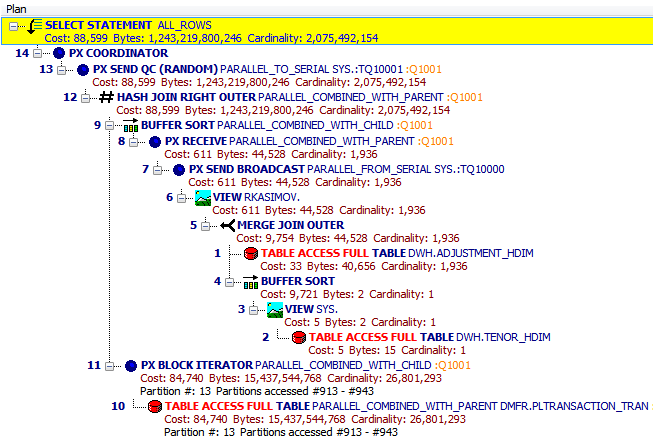
left join dmfr.tenor\_ldim t on

t.ccode = 'Y25'

) adj on

pl.adjustment\_mark\_uk = adj.uk

where pl.value\_day between '01.07.2013' and '31.07.2013'



Видим, что при таком подходе проводки уже не сортируются, а сортируются два маленьких справочника. Такой запрос отработает гораздо быстрее предыдущего. Правда стоит обратить внимание, что на выходе получили некорректную кардинальность, что может повлиять на план выполнения запроса, если запрос не ограничивается этими тремя таблицами. Но для нашего примера это неважно.

1. Использование инициализирующей и инкрементальной загрузок в одном запросе.

Зачастую в ETL используется в sq один запрос, который через union all объединяет две части – инициализирующую (которая в инкрементальной загрузке получает фильтрацию 1=0) и инкрементальную. Но зачастую вопреки ожиданиям выполняются оба подзапроса, что сильно нагружает базу.

*Пример.*

select */\*+parallel(d,2) parallel(d2,4)\*/*

d.nk, d2.\*

from

DWS024\_DEBTM.KHDEBTP024\_DELTA d

LEFT JOIN DWS024\_DEBTM.KHDEBTP024\_MIRROR D2

ON

trim(d2.KHWTCF) = 'ICI' AND

trim(d2.KHMISV) = 'ND' and

D.KHP1ND = D2.KHP1ND

where

0 = 0

union all

select */\*+parallel(d,4) parallel(d2,4)\*/*

d.nk, d2.\*

from

DWS024\_DEBTM.KHDEBTP024\_MIRROR d

LEFT JOIN DWS024\_DEBTM.KHDEBTP024\_MIRROR D2

ON

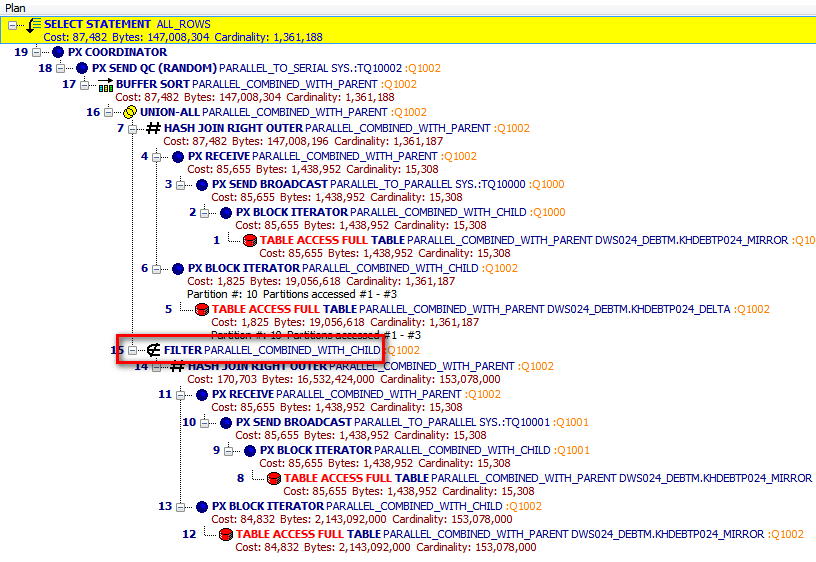
trim(D2.KHWTCF) = 'ICI' AND

trim(D2.KHMISV) = 'ND' and

D.KHP1ND = D2.KHP1ND

where

0 = 1

       Фильтрация 1=0 выполняется после джойна таблиц, а не до как мы того ожидали! Такие запросы просто стоит выполнять отдельно.

1. Пример пошагового чтения плана и оптимизации запроса

Преположим у нас есть такой запрос, план выполнения которого приведен ниже:

select

CLIENT\_PIN, ACCNUM, Start\_date, INN, COMPANY, Agent\_PIN, Agent\_name, ChUK, Address,Branch

from

(

select

CS.Client\_Pin CLIENT\_PIN, ACS.Account\_Number ACCNUM, ACS.START\_DATE Start\_date, CS2.Taxpayer\_Ncode INN, CS2.Client\_Name COMPANY, CS3.Client\_Pin Agent\_PIN, cs3.client\_name Agent\_name, CSD.CCODE ChUK, ADS.Name Address, SPS.Name Branch

,row\_number() over (partition by acs.start\_date, cs.client\_pin, acs.account\_number, cs2.taxpayer\_ncode order by cs3.client\_pin, cs2.taxpayer\_ncode) N

from dmrb.ACCOUNT\_SDIM ACS

inner join dmrb.CLNT2DEALSALARY\_SHIST CSH on CSH.Client\_Uk=ACS.Client\_Uk

left join dmacrm.client\_sdim CS on CS.Uk=CSH.Client\_Uk

left join dmrb.dealsalary\_sdim DS on DS.UK=CSH.Dealsalary\_Uk

left join dmacrm.client\_sdim CS2 on CS2.Uk=DS.Client\_Uk

left join dmacrm.client\_sdim CS3 on CS3.Uk=CSH.Client\_Agent\_Uk

left join dmrb.Slragentparam\_Shist SAS on SAS.Client\_Agent\_Uk=CSH.Client\_Agent\_UK

left join dmrb.ADDRREF\_SDIM ADS on ADS.UK=SAS.Addrref\_Region\_Uk

left join dmrb.channel\_sdim CSD on CSD.UK=DS.Channel\_Uk

left join dmrb.SALESPLACE\_SDIM SPS on SPS.UK = CS3.Salesplace\_Origin\_Uk

where

ACS.START\_DATE between to\_date('20140101','yyyymmdd') and to\_date('20140131','yyyymmdd')

and (acs.accounteqtype\_uk=5212351361 or acs.accounteqtype\_uk=5212350214)

and acs.deleted\_flag='N'

and csh.deleted\_flag='N'

and csh.effective\_to >= ACS.START\_DATE

and cs.deleted\_flag='N'

AND (ds.end\_date>= ACS.START\_DATE or (DS.uk IS NULL AND CSH.Dealsalary\_Uk is not null))

and (ds.deleted\_flag='N' or (DS.uk IS NULL AND CSH.Dealsalary\_Uk is not null) )

AND (CS2.Deleted\_Flag='N' or (DS.uk IS NULL AND CSH.Dealsalary\_Uk is not null))

AND (CS2.Taxpayer\_Ncode is not null or (DS.uk IS NULL AND CSH.Dealsalary\_Uk is not null))

AND (CS3.DELETED\_FLAG='N' or cs3.uk is null)

AND (SAS.Deleted\_Flag='N' or SAS.Client\_Agent\_Uk is NULL)

AND

(

SAS.effective\_to >= ACS.START\_DATE

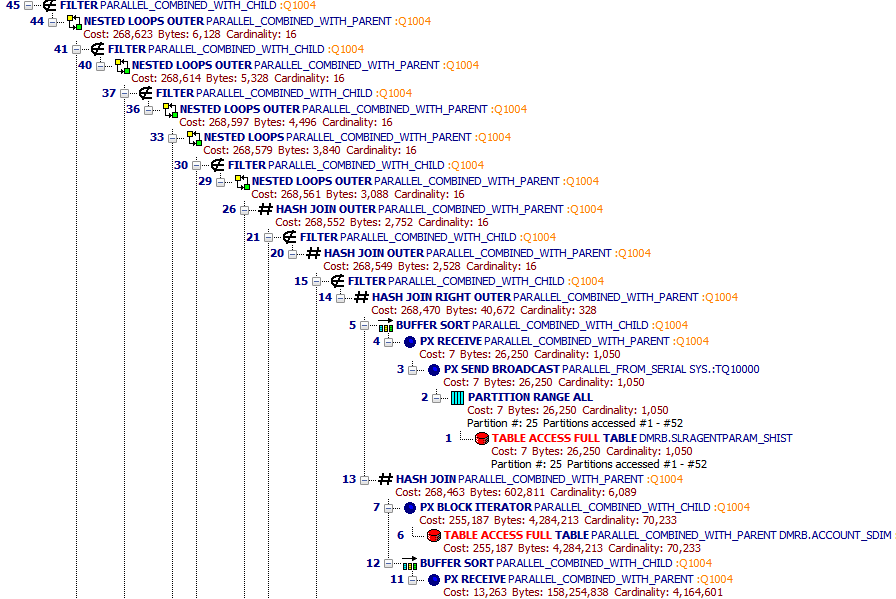
or SAS.Client\_Agent\_Uk IS NULL

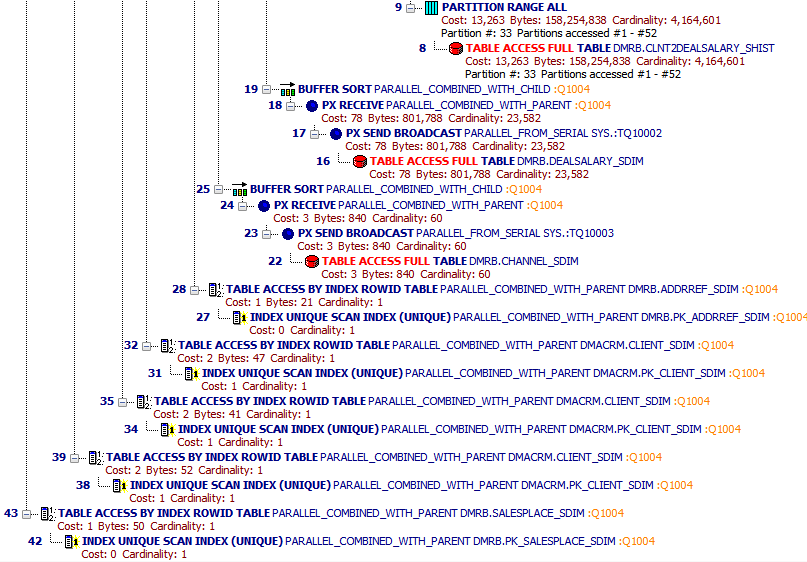
)

AND (ADS.Deleted\_Flag='N' or ADS.Uk is null)

AND (SPS.Deleted\_Flag='N' or sps.uk is null)

) t where N=1





1) Выполнение запроса начинается с чтения и хэширования таблицы DMRB.SLRAGENTPARAM\_SHIST (см. шаг 1 в плане). Таблица эта небольшая (исходная, а не по оценке оптимизатора с учетом фильтрации!), поэтому этот шаг нам неинтересен, пропускаем его.

2) Чтение и хэширование DMRB.ACCOUNT\_SDIM. Вот здесь потенциально опасное место – если оптимизатор ошибся с кардинальностью (по его оценке записей 70233) из-за фильтрации, то уже здесь может начаться просадка производительности, так как таблица счетов исходная очень большая и если на хэширование пойдет, скажем 70 млн строк, то это гарантированное использование TEMP. Поэтому проверяем. Но проверка показала, что оптимизатор не ошибся, счетов действительно отобралось 70 тыс, то есть здесь порядок соединения верный.

3) Чтение таблицы DMRB.CLNT2DEALSALARY\_SHIST – таблица исходная маленькая, заходим на нее через хэш, поэтому не обращаем на нее внимания (пока что) и идем дальше.

4) После соединения этих трех таблиц (см. в плане п.14) видим, что кардинальность соединения 328, а после соединения с DMRB.DEALSALARY\_SDIM остается 16. Естественно, что при такой низкой кардинальности оптимизатор выбирает зачастую как метод соединения со следующими таблицами вложенный цикл. Но у нас в главной таблице DMRB.ACCOUNT\_SDIM (чтобы убедиться что она главная можно посмотреть запрос) исходная кардинальность была 70 тыс строк и совсем не факт, что после нескольких соединений она стала 16 строк. Это обязательно нужно проверить. Проверка показала, что действительно оптимизатор ошибся и кардинальность вместо 16 строк 46 тыс строк! Поэтому весь дальнейший план после п.15 является неоптимальным и дальше его уже можно не смотреть, пока не исправим кардинальность до этого момента.

Варианта исправления здесь два:

1. либо мы накладываем хинтом повышенную кардинальность на таблицу DMRB.ACCOUNT\_SDIM и другие первые таблицы, чтобы после соединения с DMRB.DEALSALARY\_SDIM в плане была кардинальность примерно 46 тыс
2. либо хинтами use\_hash, leading жестко прописываем весь план

К данному случаю не относится, но напомню, что стоит быть аккуратным с хинтами, так как при этом может измениться структура плана так, что хэш-функция будет строиться уже по большому объему данных. Этого нужно обязательно избегать!

И еще стоит отметить – с кардинальностью 46 тыс на все маленькие справочники необходимо идти с хэш-соединением. Но есть и достаточно крупные – DMRB.CLIENT\_SDIM. Как будет оптимальнее с ним соединиться при такой кардинальности, покажет только эксперимент, так как здесь можно ошибиться.

# Заключение

В двух обучающих курсах было изложено много теоретического материала и приведено достаточное количество примеров, которые позволят избежать подавляющее большинство неоптимальностей при разработке и просто при написании запросов.

Подводя резюме что же необходимо для оптимизации запросов:

1. Знание теоретических основ, которые как раз изложены в обучающих курсах
2. Опыт, который начинается с приведенных примеров и далее увеличивается по мере того как приходится решать подобные проблемы

Основные правила при работе со сложными запросами:

1. В первую очередь необходимо обратить внимание на чтение больших таблиц – правильный ли метод доступа выбран – полное сканирование или индексный доступ, если индексный то оптимальный ли индекс. Максимально ли ограничено чтение таблицы по партициям
2. Если есть сомнения в методе доступа из-за кардинальности, то нужно выполнить запрос select count(\*) до этой таблицы, получив истинную кардинальность. Далее уже при необходимости хинтами исправлять план запроса
3. В случае очень сложного запроса разбиваем на куски, проверяя с самого начала и останавливаясь на тех моментах, где план может быть неоптимальный
4. Крайне желательно избегать сортировок и построение хэш-функций (напоминаю, что первой в хэш-соединении должна идти меньшая таблица) по большим объемам.
5. Необходимо избавляться от коррелированных запросов, то есть таких, которые выполняются для каждой строки в селекте
6. Также нужно избегать многократного выполнения одной операции, например, когда миллионы строк идут на соединение вложенным циклом
7. Необходимо стараться избегать размножения данных при соединениях и других операциях
8. Чем раньше и сильнее будет уменьшен объем данных, тем лучше.