УДК 004.655

ОПТИМИЗАЦИЯ ЗАПРОСОВ В GREENPLUM

Павлович Н.В.

Белорусский государственный университет информатики и радиоэлектроники, г. Минск, Республика Беларусь

Научный руководитель: Тонкович И.Н. – канд.хим.наук, доцент, доцент кафедры ПИКС

Аннотация. В статье рассматриваются основные понятия и механизмы оптимизации запросов в Greenplum. Выявлены причины неоптимальной работы запросов и приводятся рекомендации по решению данной проблемы.

Ключевые слова: Greenplum, оптимизация, план запроса, перекос, движение данных, замножение данных

Введение. Уже много лет существует тенденция, при которой для крупного бизнеса возрастает значимость распределенных аналитических баз данных, таких как *Greenplum*. Поэтому задача оптимизации запросов стоит как никогда остро. При плохо составленных запросах скорость обработки любого запроса равна скорости самого медленного сегмента [1].

В статье рассматриваются основные причины неоптимальной работы запросов и даются рекомендации по решению данной проблемы.

Основная часть. При медленной обработке запроса необходимо, во-первых, понимать причину, а во-вторых, найти ее решение. Для установления причины следует изучить план запроса. План запроса – это последовательность операций, которые выполняет БД для получения требуемого в запросе результата. За построение плана запроса отвечает оптимизатор – модуль БД, который учитывает такие факторы, как количество записей в таблицах, наличие индексов, наличие партиций [2]. План запроса представляет собой дерево, где каждый узел – какая-либо операция БД.

Выделим основные операции, которые используются при прочтении плана запроса.

Операции движения данных:

1 *Redistribute motion* – перераспределение данных. При перераспределении на каждом сегменте происходит отправка строк таблицы на другие сегменты в соответствии с новым ключом распределения.

2 *Broadcast motion* – полное копирование таблицы на все сегменты. Каждый сегмент посылает свою порцию таблицы на все остальные сегменты. Вследствие чего на каждом сегменте оказывается полная копия таблицы.

3 *Gather motion* – собирает результат работы запроса со всех сегментов.

Операции соединения:

1. *Hash join* – наиболее часто встречающийся тип соединения таблиц в *GP* и в подавляющем большинстве наиболее оптимальный из всех. Алгоритм соединения заключается в чтении меньшей таблицы и построении для нее хэш-таблицы, далее читается большая таблица, для каждой строки которой вычисляется значение хэш-функции от ключа соединения, и выполняется поиск этого значения в хэш-таблице. В случае успеха соответствующие записи из обоих таблиц добавляются к результату соединения. Указанные действия выполняются параллельно на всех сегментах, поэтому необходимо, чтобы на момент выполнения *hash join* соединяемые таблицы были распределены по ключу соединения.

Соединение двух таблиц с помощью *nested loop* выполняется путем прохода по всем строкам меньшей таблицы для каждой строки большей таблицы. В случае выполнения условия соединения соответствующая пара строк попадает в результирующую таблицу. Для выполнения *nested loop* меньшая таблица копируется (*broadcast motion*) на все сегменты. При этом меньшая таблица не всегда является маленькой. *Nested loop* является крайне неоптимальным способом соединения в случае, когда обе таблицы большие. Этот способ соединения выбирается оптимизатором при условии соединения отличном от обычного

равенства полей.

2. *Merge join* – соединение предварительно отсортированных таблиц. Происходит за один проход по каждой из таблиц. Для вычисления *merge join* необходима предварительная сортировка данных, а сортировка – очень тяжелая операция. Для того, чтобы увидеть план запроса нужно использовать команду *Explain* перед запросом. В строке помимо типа выполняемой операции содержатся следующие метрики: *cost* – стоимость выполнения операции, измеряется в единицах чтения дисковых страниц; *rows* – итоговое количество строк, которое в среднем возвращает каждый сегмент после выполнения данного оператора; *width* – ширина выходной таблицы в байтах.

В общем случае существует множество вариантов для выполнения одного запроса, и задача оптимизатора выбрать наиболее оптимальный. Для того, чтобы выбрать наиболее оптимальный вариант, оптимизатор должен иметь статистику о таблицах, которые участвуют в запросе. В статистике для таблицы содержится информация об объеме таблицы, для каждого поля таблицы указывается информация о количестве *null*-значений, количестве уникальных значений, наиболее часто встречающиеся значения и т.д. [3].

В случае, когда статистика на таблицу неактуальная, т.е. после последнего сбора статистики данные в таблице сильно изменились, оптимизатор может выбирать неоптимальный план запроса. В *GP* есть два варианта выбора оптимизатора: *GPORCA* и *Legacy optimizer*. По умолчанию используется оптимизатор *GPORCA*, который является по сути улучшенной и доработанной версией оптимизатора *Legacy optimizer*.

Greenplum работает со скоростью самого медленного сегмента. Какой же сегмент является самым медленным? Мощность каждого сегмента одинакова и операции на каждом сегменте также выполняются одинаково. Что может отличаться, так это объем данных, который обрабатывается каждым сегментом. Поэтому самым медленным сегментом оказывается тот, который обрабатывает наибольшее количество строк. В идеальной ситуации, при которой все таблицы и промежуточные результаты запроса распределяются равномерно по всем сегментам, нагрузка на все сегменты одинакова и запрос работает наиболее оптимально. Но в реальности случается так, что-либо исходные таблицы, либо промежуточные результаты запроса распределяются таким образом, что на одном или нескольких сегментах оказывается большая часть таблицы [4]. Именно в этом случае принято говорить о наличии перекоса в данных.

Выделяют следующие варианты перекоса: перекос из-за *null*-значений в ключе распределения и перекос из-за плохой уникальности ключа распределения. Места перекоса: перекос исходных таблиц запроса; перекос при перераспределении исходных таблиц и перекос при перераспределении промежуточных результатов запроса.

Как правило, перекос находят с помощью плана запроса по следующему алгоритму:

– определяют, есть ли в плане запроса перераспределения;

– если нет, то осуществляют поиск в исходных таблицах;

– если да, то для каждого перераспределения необходимо посмотреть, что перераспределяется (исходные таблицы или результат каких-то промежуточных вычислений); как распределяется по сегментам (есть ли в ключе распределения *null*-значения или просто одинаковые значения в большей части таблицы).

Общий алгоритм действий для устранения перекоса следующий:

1 Если много уникальных значений в ключе распределения, то сначала следует выделить из таблицы ту часть строк, из-за которой возникает перекос (при этом необходимо распределить эту часть по уникальному ключу). Затем выполняют необходимые вычисления отдельно для этих частей (при этом важно не допускать перераспределения той части, из-за которой возникал перекос). И далее – объединить результаты вычислений.

2 Если мало уникальных значений в ключе распределения (и если характер вычислений позволяет работать только с уникальными ключами), то сначала группируются все записи по ключу распределения, выполняются вычисления. Далее выполняется присоединение обратно к исходной таблице через *broadcast* (т.е. важно чтобы выполнялось именно копирование

маленькой таблицы, вместо перераспределения изначальной, в противном случае проблема не решается).

В целом можно сказать, что любая операция движения данных негативно сказывается на времени работы запроса. В идеальном случае запрос должен выполняться параллельно на всех сегментах без каких-либо движений, данных между ними. Но в реальности таблицы могут использоваться в совершенно разных запросах, поэтому нельзя придумать такое распределение таблицы, чтобы в каждом запросе не было никаких движений данных.

Тем не менее в случаях, когда движения данных необходимы, у оптимизатора часто имеется несколько разных вариантов движения исходных таблиц. Например, при соединении двух таблиц по ключу, отличному от ключей распределения исходных таблиц, у оптимизатора есть следующие варианты: перераспределить обе таблицы по ключу соединения или скопировать одну из таблиц на все сегменты, а вторую оставить как есть.

Задача оптимизатора при построении плана запроса – выбрать среди всех вариантов тот, при котором объем перемещаемых между сегментами данных будет минимальным. Но иногда случается так, что оптимизатор выбирает не самый оптимальный вариант. Например, вместо перераспределения двух равных по объему таблиц, он решает одну из них скопировать на все сегменты или наоборот, вместо копирования маленькой таблицы, перераспределяет огромную.

Это может происходить по следующим причинам:

1 Неактуальная статистика на таблицы-источники. В этом случае оптимизатор просто может не знать о реальных объемах таблиц, в следствие чего ошибиться в выборе варианта движения данных. Понять, что статистика на таблицу неактуальна можно по плану запроса.

2 Разные типы ключей соединяемых таблиц. В случае соединения двух таблиц по полю, тип которого отличается в исходных таблицах, может происходить перераспределение одной из исходных таблиц.

3 Особенности работы запроса. Например, в случае выполнения *left join* двух таблиц никогда не будет происходить копирование левой таблицы, независимо от ее объема и объема правой таблицы.

Чтобы понять, что оптимизатор делает что-то неверно, следует, во-первых, прочитать план запроса и понять какие движения данных выполняются, во-вторых, определить есть ли другие варианты движения данных и, в-третьих, сравнить все варианты на объем перемещаемых данных.

Проблема замножения данных с одной стороны является одной из самых простых, так как ее легко диагностировать, а с другой – самой сложной, поскольку решение этой проблемы в каждом случае индивидуально. Под замножением данных понимается ситуация, при которой в результате операции *join* объем обрабатываемых данных резко возрастает. Происходит это по причине наличия одинаковых значений в ключе соединения обеих таблиц.

Устранение проблем с замножением данных:

1 Если выполняется соединение таблиц по равенству каких-то полей, необходимо проверить наличие дублей по этому полю. Если нашлись дубли, то необходимо понять, нужны ли все эти данные в результирующей таблице. Когда они не нужны, то необходимо дедублицировать (сгруппировать или выбрать какую-то одну строку для каждого значения) исходные таблицы по ключу соединения.

В случае, когда они нужны только для вычислений (т.е. будет выполняться какая-то агрегация этих значений), то можно попробовать предагрегировать хотя бы одну из исходных таблиц, сделав необходимые вычисления и избавившись от дублей по ключу соединения, а затем соединить таблицы (уже без замножения) и доагрегировать получившиеся данные.

2 Если выполняется соединение таблиц не только по равенству полей, т. е. присутствуют какие-то дополнительные условия (отличные от равенства полей), то необходимо проверить на наличие дублей те поля, которые участвуют в условиях на равенство. Оценить объем замножения в этом случае можно, соединив таблицы только по условиям на равенство.

Если объем замножения является огромным по отношению к исходным таблицам (в сотни раз больше), то необходимо попробовать добавить в условие соединения еще одно или несколько условий по равенству полей так, чтобы вместе с ними замножение значительно уменьшилось.

Далее опишем общие рекомендации по написанию запросов в *Greenplum*, что позволит в большинстве случаев избежать ошибок, рассмотренных ранее [5]:

– в идеальном случае join следует делать по полям дистрибуции хотя бы одной из таблиц;

– необходимо максимально уменьшать количество данных перед операцией *join*;

– рекомендуется всегда указывать только необходимые колонки в итоговом *select*-запросе;

– использовать партиции. Одним из главных механизмов оптимизации запросов к большим таблицам в *GP* является партиционирование;

– не используем or, in, *like* в условиях соединения по полям;

– операцию сортировки не стоит ее использовать без крайней необходимости;

– не рекомендуется использовать оператор *Distinct*. Он часто требует перераспределения таблицы по всем указанным в запросе полям;

– не рекомендуется использовать оператор *Union.*

Для объединения результатов запросов рекомендуется использовать *Union All.* Оператор *Union*, кроме объединения результатов запросов, удаляет дубли. При этом для этого GP требуется выполнить одну из самых тяжелых операций – сортировку.

Заключение. Практическое применение наработанных идей позволяет в значительной степени повысить производительность распределенной базы данных. Что, в свою очередь, снизит потребность в приобретении дополнительных производительных мощностей.

Список литературы

1. *Open Source Greenplum Database [Электронный ресурс]. – Режим доступа: https://docs.greenplum.org/6-16/common/gpdb-features.html.*

2. *Планы SQL-запросов в Greenplum [Электронный ресурс]. – Режим доступа: https://www.bigdataschool.ru/blog/explain-sql-queries-in-greenplum.html.*

3. *Сборщик статистики [Электронный ресурс]. – Режим доступа: https://postgrespro.ru/docs/postgresql/9.6/monitoring-stats.*

4. *Перекос [Электронный ресурс]. – Режим доступа: https://www.bigdataschool.ru/blog/greenplum-data-and-process-skew-how-to-find-and-solve.html.*

5. *Оптимизации SQL-запросов в Greenplum [Электронный ресурс]. – Режим доступа: https://habr.com/ru/company/rostelecom/blog/442758/.*

UDC 004.655

OPTIMIZING QUERY IN GREENPLUM

Paulovich N.V.

Belarusian State University of Informatics and Radioelectronics, Minsk, Republic of Belarus

Tonkovich I.N. – PhD, assistant professor, associate professor of the department of ICSD

Annotation. The article discusses the basic concepts and mechanisms of query optimization in Greenplum. The reasons for the non-optimal work of queries are identified and recommendations are given for solving this problem.

Keywords: Greenplum, optimization, query plan, skew, data movement, data multiplication