УДК 004.657:681.3

ОПТИМИЗАЦИЯ ЗАПРОСОВ В СИСТЕМАХ БАЗ ДАННЫХ НА ПАРАЛЛЕЛЬНЫХ СТРУКТУРАХ

Д.И. Щетинин

Институт кибернетики им. В.М. Глушкова НАН Украины
03680, Киев, проспект Академика Глушкова, 40,
т.: +38(044) 526-15-05; e-mail: mdg@dmipt.icyb.kiev.ua

В работе представлены наиболее распространенные виды параллельных реляционных СУБД. Определяются понятия времени выполнения запроса (времени отклика системы на запрос) в параллельной системе. Предлагаются эффективные способы вычисления времени отклика.

The most widespread types of parallel relational DBMS are presented in paper. The notions of query execution time (response time) are determined in the parallel system. The effective methods of response time calculation are offered.

Введение

В работе рассмотрены вопросы, связанные с оптимизацией выполнения запросов в распределенных реляционных системах управления базами данных (СУБД). Рассмотрены некоторые открытые вопросы, оптимизации времени обращения к устройствам хранения данных в СУБД и времени передачи данных по сети. В работе описывается архитектура такой СУБД, ее общая логика работы, адресные и логические пространства данных, способы доступа к этим данным в архивах СУБД. Рассматриваются основные проблемы оптимизации, построена модель, на базе которой вводятся функции оценок и области определения этих функций. В работе показано, как поступающий в СУБД SQL запрос разбивается на множество подветвей, образуя, таким образом, операторное дерево. На основе такой модели вводится понятие компонента СУБД – оптимизатора. В самой общей постановке, задачей оптимизатора является генерация множества альтернативных планов выполнения данного запроса и выбора среди них плана с наименьшим временем выполнения. Рассматривается подход к обработке глобальных запросов, основанный на их компиляции. Такие способы были предложены в System R [1] и использовались в System R* [2].

1. Архитектура распределенных СУБД

Существуют различные способы построения высокоскоростных параллельных систем управления данными, а именно, СУБД с совместно используемой основной памятью, sharing-memory (shared everything, tightly coupled [2]), СУБД с совместно используемым дисковым пространством, shared-disk (data-sharing, loosely coupled) и СУБД без совместного использования памяти и дискового пространства, shared-nothing (partitioned data).

В системах с общей памятью процессоры используют общую центральную память [3]. В системах с совместным дисковым пространством, процессоры, каждый из которых имеет свою локальную память, используют общий дисковый пул. В shared-nothing системах каждый процессор имеет свою локальную память и свое локальное дисковое пространство. Детальное описание архитектур распределенных СУБД представлено в [2]. Архитектура DB2 Parallel, а также способы выполнения и обработки запросов представлены в [2] и [4].

Следует заметить, что в последнее время активно развивается проект MySQL Cluster. Физическая структура, а также способы выполнения запросов в этой среде подобны аналогичным в DB2. Более полные сведения о MySQL Cluster представлены в [5].

Основной сложностью в работе и поддержке архитектуры shared-nothing являются требования распределения запроса, языка запросов SQL, на несколько подзапросов, отправляемых к различным узлам системы, и дальнейшее слияние результатов этих запросов полученных от каждого узла системы. В то время как системы с общей памятью и общим дисковым пространством ограниченны пропускной способностью памяти или полосой пропускания, система shared-nothing может быть легко распределена на сотни процессоров, не обязательно находящихся в одной локальной сети. На рис.1 показаны три представленные архитектуры.

2. Оптимизация времени отклика на запрос в параллельной СУБД

2.1. Построение моделей оценок времени выполнения запросов в СУБД. Одной из ключевых проблем в разработке оптимизатора запросов для параллельных распределенных СУБД, является построение эффективных и точных моделей оценок времени выполнения запросов в СУБД.. Эти модели основаны на

точных оценках времени выполнения запроса, которые могут быть получены при использовании дорогостоящих алгоритмов-планировщиков. Для достижения эффективности модели оценок стоимостей должны оценивать время отклика (response time), используя функции приближения. Далее представлены основные термины, понятия, алгоритмы и прочие сущности, которые лежат в основе проектирования моделей стоимостей.

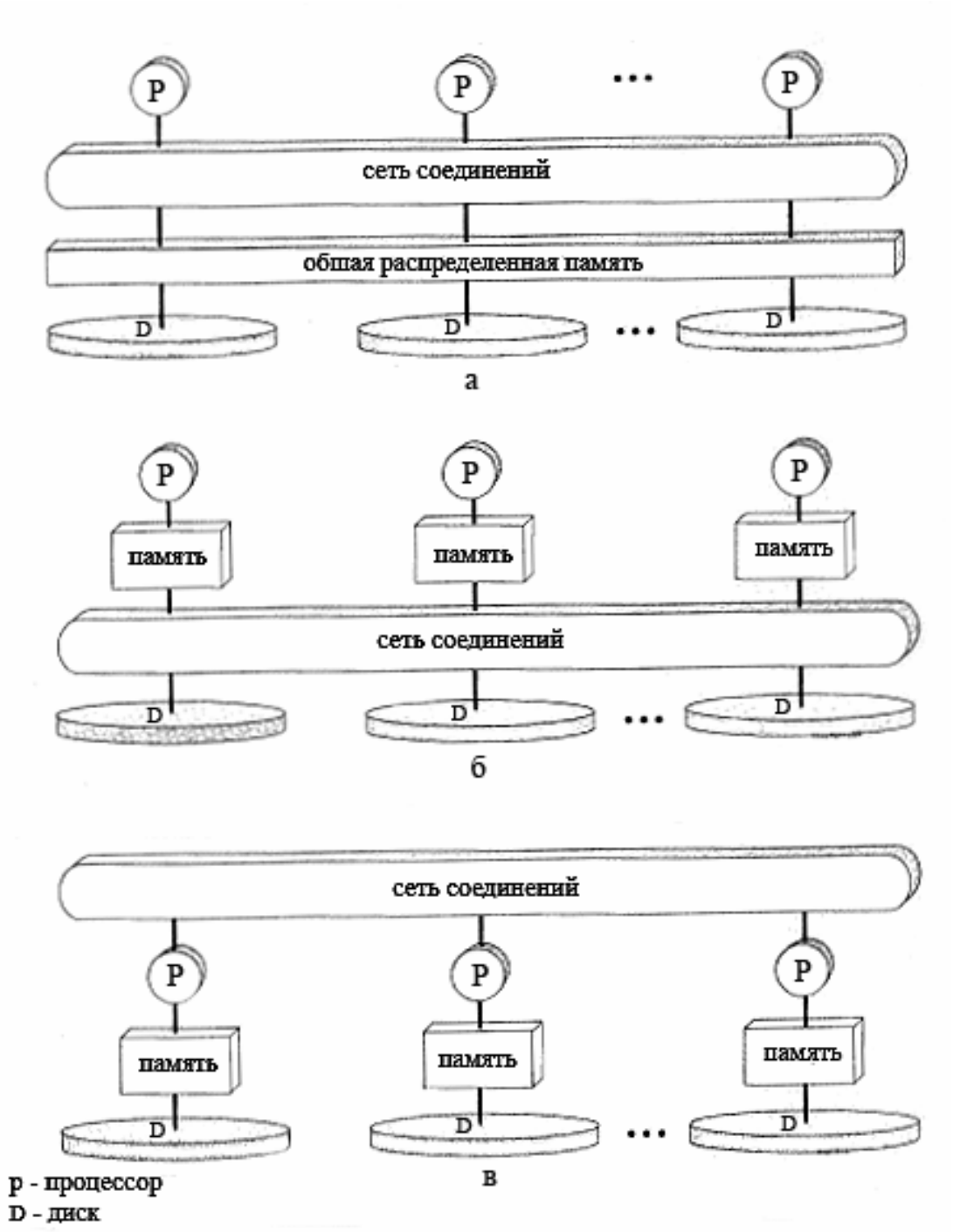

Рис. 1. Архитектуры систем: а – система с распределенной памятью; б – система с распределенным дисковым пространством; в – система не использующая распределение памяти и дискового пространства

Главной целью оптимизатора запросов в параллельной реляционной СУБД является построение такого плана выполнения данного запроса, который уменьшит время выполнения (отклика) запроса в системе. Время выполнения (отклика) для данного плана является временем, которое тратит система, выполняя параллельно этот запрос на некоторых своих узлах.

Для данного запроса существует большое число различных планов реализации, которые эквивалентны между собой; т.е. они порождают один и тот же ответ на запрос, определяемый семантикой SQL. Среди множества всех планов исполнения существует план исполнения, который имеет наименьшее время ответа (отклика). Цель оптимизатора – нахождение этого оптимального плана исполнения. Будем использовать модель

стоимостей (cost model) для оценки времени выполнения запроса по данному плану и для поиска пространства исполняемых планов при условии, что данный план имеет малое значение стоимости. Точность процедуры поиска и ее сложность будут зависеть от точности самой модели стоимостей и вычислительной сложности модели стоимостей.

Проблема оптимизации запросов в параллельной СУБД, где запросы выполняются последовательно, каждый в свое время, может быть описана следующим образом. Оптимизатор получает на вход обычный SQL запрос, спецификацию архитектуры системы, в которой выполняется этот параллельный запрос и логическую структуру данных в СУБД. На выходе оптимизатор предоставляет оптимальный план исполнения запроса.

Прямой путь решения этой проблемы – множество последовательных планов исполнения данного запроса, применимых к запросам в классической СУБД, в которой не существует параллельного выполнения запросов преобразуется во множество параллельных планов исполнения, применимых к запросам в параллельных СУБД. Это можно выполнить, определив местонахождение ресурсов в соответствии с каждой операцией в последовательном плане исполнения и определив время начала каждой операции. Таким образом, можно представить параллельный план исполнения как некое декодирование последовательного плана исполнения так, чтобы, в зависимости от расписания, выполнять каждую операцию в этом плане на отдельной параллельной машине. Параллельный план исполнения = последовательный план + начальные метаданные + информация о времени выполнения каждой операции. Все это делает задачу оптимизации запросов в параллельных СУБД более сложной, чем соответствующая задача оптимизации в обычных, непараллельных СУБД.

Предположим, что имеется алгоритм *A*, входом которого является последовательный план выполнения и описание параллельной машины, а выходом - время, затраченное на выполнение задачи с наименьшим временем ответа для этого плана, среди всех возможных параллельных выполнений. Используя алгоритм *A* [6] можно полностью найти пространство всех последовательных планов и потом среди них найти наилучший план.

Хотя этот подход обеспечивает оптимальный план выполнения, он не является жизнеспособным, пока не будет найден алгоритм *A*, выполняющийся за полиномиальное время. Проблема определения точности давно была сведена к использованию полиномиального алгоритма *B* [6], который предназначен для аппроксимации оценки полученной алгоритмом *A*. Различные подходы нахождения алгоритма *B*, используемые в литературе, можно разбить на два основных класса:

Алгоритмы перегрузки функций, точность которых зависит только от теоретических аргументов или экспериментальных утверждений.

Оценка минимального время обработки последовательного плана на параллельной машине с использованием распределителя ресурсов. Распределитель ресурсов принимает последовательный план (вместе с оценками времени выполнения последовательных операторов, информации о данных, размерами промежуточных значений и т.д.) и строит некое задание параллельного выполнения данного плана выполнения. Достижение времени завершения задания рассчитывается и используется как аппроксимация выходных значений алгоритма *A*.

У обоих подходов есть ряд недостатков. В первом подходе точность оценочных функций не достаточно правомерна. Второй подход более устойчив, хотя с вычислительной точки зрения более дорогой. Алгоритмы распределения ресурсов для последовательных планов, как известно, имеют сложность порядка $O(n \cdot m)$ или выше, где n – число ресурсов, доступных параллельной машине, и m – размер последовательного плана выполнения. Если бы оптимизировалась последовательная СУБД, то время, необходимое для выполнения запроса было бы $O(m)$. В случае параллельной СУБД число n может принимать значения сотен и тысяч. Таким образом, время, требуемое для оптимизации запроса в параллельной СУБД может потенциально иметь величину нескольких порядков и быть намного больше времени, требуемого для оптимизации последовательного выполнения.

В работе предложен альтернативный подход оценки минимального времени выполнения параллельного запроса для данного исполняемого плана, который позволяет оценить время выполнения, используя функции дешевых оценок, не создавая расписание для размещения ресурсов последовательного плана. Эти функции оценок позволяют производить более точные расчеты, чем алгоритмы распределения ресурсов. Данный подход позволяет:

– построить методы оптимизации параллельных запросов, сложность которых сравнима с оптимизацией последовательных запросов;

– создать планы параллельного выполнения запросов, которые являются оптимальными планам (в терминах времени выполнения), с большой вероятностью.

На экспериментальных данных показано, что функции оценок стоимостей строят планы выполнения, которые для 95 % запросов на 20 – 60 % более оптимальны по сравнению с планами, предоставляемыми функциями построения точного расписания . Это значит, что если функции оценок используются для оценивания времени выполнения последовательного плана на параллельной машине, то для 95 % запросов наилучший план будет иметь время выполнения, отличающееся от времени даваемого функциями точного расписания не больше чем в 1.2 – 1.6 раза.

Для определения модели оценок, будем использовать понятие *операторного дерева*, как дополнение к понятию исполняемое дерево, в котором каждый из узлов представляет определенный метод слияния операндов и отношений доступа. Каждая операция слияния может быть композицией нескольких операций; например, сортирования и слияния файлов, построения хеш-таблицы и ее обработки. Операторное дерево представляет собой разбиение каждого узла исполняемого дерева на его составляющие операторы. Зависимости данных между соседними операциями в операторном дереве размещается на краях дерева. Будем полагать существование двух зависимостей данных между операциями: *последовательная* и *конвейерная*. Последовательная зависимость возникает в ситуациях, когда потребитель должен ждать, пока поставщик закончит выполнение. Например, когда поставщик или потребитель является операцией “sort”, или когда поставщик является “build-hash-table” операцией и потребитель является “probe-hash-table” операцией. Конвейерная зависимость возникает в ситуациях, когда потребитель может начать использовать ресурсы до того, как поставщик закончит выполнение. Например, когда поставщик является операцией “merge”, или когда поставщик – операция “probe-hash-table”, а получатель – операции “index-scan” по “nested-loop”. Достаточно легко построить таблицу зависимостей данных среди всех пар реляционных операторов.

Считаем, что рассматриваемая модель является централизованной моделью оценок стоимостей, которая оценивает:

– время работы каждого оператора в операторном дереве в последовательной машине;

– размеры входных и выходных данных для каждого оператора;

– потребность выделения памяти для работы каждого оператора.

Предположим, что система основана на архитектуре shared-nothing, состоит из множества процессоров $P = \{P_1, P_2, \ldots, P_n\}$, соединенных друг с другом по сети. Каждый P_i соединен с локальным разделом хранения данных (т.е. обычный диск, несколько дисков или RAID массив). Считаем процессоры и диски идентичными. Операторы могут быть распараллелены; т.е. их можно выполнять параллельно, используя несколько процессоров, а операторы relational-scan не могут быть произвольно распараллелены, поскольку они считывают данные с дисков.

Рассмотрим такое операторное дерево, в котором выходные значения оператора o_i используются оператором o_j. Предположим, что для выполнения операторов o_i и o_j используются множества процессоров Q_i и Q_j соответственно. Полагаем, что $send_{ij}$ соответствует величине затрат (стоимости), производимых каждым процессором из Q_i (исходя из предположений об их симметрии, считаем что они эквивалентны между собой) при отправке своих выходных результатов множеству процессоров Q_i. Аналогично, полагаем, что $receive_{ij}$ соответствует величине затрат, производимых каждым процессором из Q_j (в предположении, что эти процессоры эквивалентны друг другу) при получении данных, отправленных процессорами множества Q_i. Термин $delay_{ij}$ соответствует времени, потраченному на отправку сообщения из какого-либо процессора множества Q_i и на доставку соответствующего сообщения процессору-получателю из множества Q_j. Положим Z_i множество процессоров, между которыми возникли связи при выполнении операции o_i и ее распределении по множеству Z_i, и пусть Z_j определяет множество процессоров получателей, по которым эти связи были перераспределены. При этом распределение отношений как в Z_i, так и в Z_j происходит по одному и тому же алгоритму.

Параллелизм распределения данных реляционных операторов производится с помощью выбора атрибута(ов) раздела (partitioning attribute). Например, предположим, что существует соединение двух отношений *R(a,b,c)* и *S(d,e,f)* с предикатом соединения *R.f = S.d*. В этом случае естественно использовать *b* как атрибут раздела отношения *R* и *d* как атрибут раздела отношения *S*. Далее выбирается хеш-функция *h* и применяется как к атрибутам *b* или *d* для каждого кортежа в отношениях *R* или *S* соответственно. Если возвращаемое значение функции равно *k*, то кортеж отправляется *k*-му процессору. Это означает, что кортежи отношений *R* и *S*, которые отправляются к соответствующим процессорам, не могут быть соединены.

Теперь предположим, что результат выше описанного соединения *R* и *S* соединяется с другим отношением *T(j,k,l)* с предикатом соединения *S.f = T.j*. В этом случае атрибутом раздела для этого соединения будет *f* для $R \infty S$, и *j* для *T*. Однако отношение $R \infty S$ уже разделено с помощью атрибута *b* (или *d*, т.к. *b = d*), и требуется перераспределение по другому атрибуту – *f*. Пусть теперь предикат соединения имеет вид *S.d = T.j*. В этом случае выходные данные отношения *S* уже распределены по атрибуту *d*. Последний из упомянутых случаев называется *один ко многим* (*one-to-few*), а первый из упомянутых называется случаем *все ко всем* (*all-to-all*).

Для объяснения этих понятий положим, что соединение $R \infty S$ выполняется на множестве процессоров $P = \{P_1, P_2, \ldots, P_k\}$, а второе соединение выполняется на множестве процессоров $P = \{P_{k+1}, P_{k+2}, \ldots, P_{2k}\}$. Далее, для *i* из $1 \le i \le k$, процессор P_i содержит все кортежи отношения $R \infty S$, в которых *h(b) = h(d) = i*. Если выбрать хеш-функцию для конечного соединения в виде $h'(x) = h(x) + k$, то в случае *один ко многим* кортежи в P_i отправляются только к P_{k+i}. В случае *все ко всем* кортежи в P_i перераспределяются с новым атрибутом хеширования *j* и отправляются, возможно, каждому к процессору в P_{i+k}, для *i* из отрезка $1 \le i \le k$.

Оценка связи при отправке простого сообщения размерностью *B* байтов от получателя, к удаленному адресату моделируется следующим образом: $t_{send} = t_{recieve} = \alpha + B\beta$, где α константа операционной системы, связанная с генерацией сообщения и процессом его отправки, β оценка стоимости передачи одного байта сообщения по каналу сетевого соединения.

2.2. Исключающая и разделенная буферизация. Пусть Q_i – множество процессоров, используемых для выполнения операции o_i, Q_j – множество процессоров, выполняющих операцию-наследник o_j. Другими словами, o_i – операция производитель, o_j – операция потребитель. В исключающей схеме буферизации предполагается, что каждый процессор в Q_i, выполняющий операцию o_i, содержит $|Q_j|$ отдельных буферов (каждый на определенный процессор в Q_j) каждый размером *B*. Когда процессор Q_i желает отправить кортеж какому-нибудь процессору P_k в Q_j, он добавляет кортеж в буфер соответствующего процессора P_k., когда буфер заполняется, он отправляется в место своего назначения. Выражения для t_{mi}^R и t_{in}^S будут представлены далее.

В схеме разделенной буферизации каждый процессор в параллельной машине, состоящей из *n* процессоров, имеет *n-1* буфер размеров *B*, по одному для каждого из оставшихся *n-1* процессоров в параллельной системе. Каждый раз, когда запрос требует отправки сообщения от процессора P_i к процессору P_j $(i \neq j)$, сообщения добавляется к *j*-му буферу процессора P_i. Отсылка буферов производится лишь после их полного заполнения.

В работах [6 – 7] стоимость передачи сообщения размера *K* байтов приблизительно оценивается как $[K / B] \cdot B \cdot \gamma$, где γ это 2β и $B \geq \alpha / \beta$.

2.3. Оценки времени отклика. Чтобы определить функции оценок *G* и *H*, необходимо сначала определить две функции *A* и *C*. Для данного операторного дерева *T*, *A(T)* определяет среднюю работу по обработке данных, производимых процессором. Функция *C(T)* (критическая длина пути) определяет длину самого длинного пути от корня к листу дерева. Элемент структуры запроса в операторном дереве показан на рис. 2.

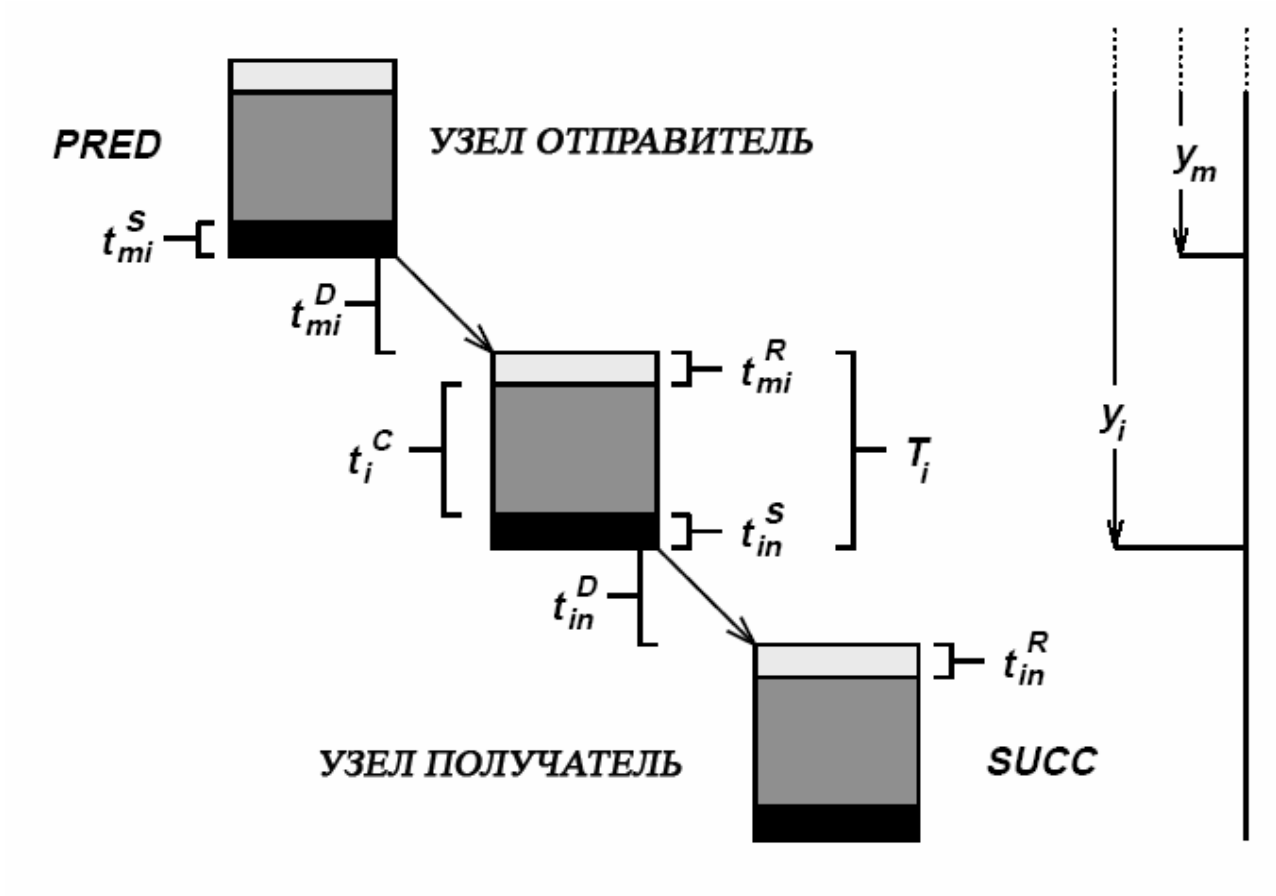

Рис. 2. Элемент структуры запроса в операторном дереве

Для последующего изложения и определения функции *A(T)* и *C(T*) введем:

– x_i – степень параллелизма узла *i* в операторном дереве;

– t_i – время индивидуальной работы каждого процессора x_i в узле *i*;

– $PRED_i$ – множество узлов отправителей узла *i* – все узлы которые отправляют данные в узел *i*;

– $SUCC_i$ – множество узлов получателей узла *i* – все узлы, которые получают данные из узла *i*;

– t_{mi}^{D} – определяет время, в течении которого сообщение находилось в сети после отправки из узла $u \in PRED_i$ в узел *i*;

– t_{mi}^{R} – определяет время, требуемое узлу *i* для обработки сообщений получаемых из узла отправителя *m*;

– t_{in}^{S} – определяет время, требуемое узлу *i* для формирования сообщения и дальнейшей его отправки узлу *n*;

– *n* – число процессоров в параллельной машине;

– *m* – число узлов в операторном дереве;

– δ – время передачи одного байта по сети.

Далее определим функции *A(T)* и *C(T)*. Пусть T_i – общая оценка времени работы узла *i* в операторном дереве, а именно оценка времени получения данных из узлов-отправителей, общее время работы узла *i* при обработке данных, время отправки обработанных и перераспределенных данных в узлы-получатели:

$$Ti = \Sigma_{u \in PRED_i} t_{mi}^{R} + t_i + \Sigma_{u \in PRED_i} t_{in}^{S}.$$

Пусть y_i – время завершения работы узла *i* в операторном дереве, т.е. y_i состоит из максимального времени завершения работы среди всех узлов-отправителей узла *i* плюс время передачи сообщения от узлов отправителей в узел *i* плюс общее время работы узла *I*:

$$y_i = \max_{u \in PRED_i} (y_u + t_{mi}^{D}) + T_i.$$

A(T) – средняя работа выполняемая одним процессором, т.е. общая работа выполненная всеми узлами разделенная на число процессоров в машине:

$$A(T) = (1/n) \cdot \sum_{i=1}^{m} T_i \cdot x_i \ .$$

C(T) – максимальное время завершения работы узлов в операторном дереве:

$$C(T) = max_{i=1}^{m} y_i \, .$$

Определим функции оценок H и G следующим образом: G(T)=max(A(T),C(T)), H(T)=A(T)+C(T).

Сформулируем ряд теорем.

Теорема 1. Пусть T – операторное дерево, выполняемое на заданном множестве из n процессоров; b – высшая грань степени параллелизма среди всех узлов операторного дерева; $R_{opt}(T)$ – время отклика (ответа на запрос) дерева T, в случае использования наилучшего алгоритма работы с ресурсами. Тогда

$$G(T) \le R_{opt}(T) \le \left(1 + \frac{n}{n-b+1}\right) \cdot G(T); \qquad 0.5 \cdot H(T) \le R_{opt} \le \left(1 + \frac{n}{n-b+1}\right) \cdot H(T).$$

Доказательство. При доказательстве будем использовать теорему 1 из [7]. Заметим, что алгоритмы, на которых построены функции *G* и *H* выполняются за время равное *O(m)*. Задача расчета оптимального времени выполнения запроса относится к классу NP-полных задач, и наиболее известные приближенные решения используют *LSA* [6] эвристику с оценкой работы приблизительно $O(n \cdot m)$. Заметим, что функции *G* и *H* предоставляют хорошие оценки времени отклика (особенно в случае $b \ge n/2$) и могут быть достаточно точно рассчитаны.

Проблему точности расчета свяжем со степенью параллелизма каждого *i*-го узла. Степень параллелизма в случае использования модели распределенной буферизации (shared buffer model) будем обозначать $x_s(i)$, а в случае использования частной буферизации (exclusive buffer model) – $x_e(i)$. Пусть B_i – общий размер входных и выходных данных узла *i* (т.е. $B_i = \sum_{j \in PRED_i} B_{ji} + \sum_{k \in SUCC_i} B_{ik}$). Оператор является *крупно-структурной* [6] единицей (*coarse-grain*) в соответствии с частной моделью буферизации, если $B_i \cdot \beta \le t_i$. В случае перехода $i \to j$ свидетельствуют, что оператор является крупно-структурной единицей, если $B_{ij} \cdot \gamma \le (t_i \cdot t_j)^{1/2}$. Операторное дерево является *крупно-структурной* единицей в соответствии с частной моделью буферизации, если каждый узел (или переход) является *крупно-структурным*. *Крупно-структурные* операторные деревья широко распространены среди всех практически используемых в СУБД операторных деревьев. Пусть L_i – число узлов-соседей узла *i*, для соединения с которыми узел *i* использует шаблон соединения *один ко всем*; N_i – число узлов соседей узла *i*, для соединения с которыми узел *i* использует шаблон *все со всеми*; *B* – размер буфера страниц процессоров. Представим формулы для расчета степени параллелизма узла *i*. Далее обсудим значение этих формул в последующих теоремах.

$$x_e(i) = \begin{cases} min((t_i + B \cdot \beta)/L_i, n) & если \quad N_i \cdot (t + B_i \cdot \beta) < L_i^2 \\ min(((t_i + B_i \cdot \beta)/N_i)^{1/2}, n) & иначе \end{cases}$$

$$x_s(i) = \begin{cases} min(\lceil (t_i/(B\cdot\gamma))^{1/2} \rceil, n) & если \quad N_i \geq 1 \\ min(\lceil t_i/(B\cdot\gamma) \rceil, n) & иначе \end{cases}$$

В дальнейших рассуждениях время задержки $B \cdot \delta$ будет игнорироваться. Рассмотрим операторное дерево, работающее на *n* процессорах. Пусть A_s^{opt} и C_s^{opt} определяют минимальную среднюю работу, выполняемую одним процессором, и минимальную длину возможного критического пути соответственно. Пусть A_s^h и C_s^h определяют среднюю работу и длину критического пути для x_s выше определенного.

Теорема 2. В соответствии с распределенной моделью буферизации для операторного дерева, являющегося крупно-структурным, справедливы соотношения: $A_s^h \leq 4 \cdot A_s^{opt}$ и $C_s^h \leq 2 \cdot C_s^{opt}$.

Определим *K*-функцию следующим образом. Для данного операторного дерева *T* $K(T) = \sum_{i=1}^{m} t_i$. Пусть K_e^{opt} – минимально возможное значение, принимаемое *K* при изменении степени параллелизма каждого оператора; A_e^{opt} – минимальное значение средней работы; A_e^h – значения функций K(T) и A(T) используя формулы для x_e.

Теорема 3. В соответствии с частной моделью буферизации, для операторного крупно-структурного дерева справедливы соотношения: $K_e^h \leq 2 \cdot K_e^{opt}$ и $A_e^h \leq 4 \cdot A_e^{opt}$.

Обе теоремы позволяют подтвердить утверждение о том, что функции оценок *G* и *H* позволяют точно оценить время отклика для данного операторного дерева. Легко видеть, что сложность обеих функций – *O(m)*, в то время как сложность наиболее известных эвристических алгоритмов, таких как *LSA* [6 – 7] является $O(n \cdot m)$. Таким образом, сложность функций *G* и *H* значительно меньше сложности *LSA*-алгоритмов.

Пусть множество $\mathrm{T} = \{T_1, T_2, \ldots, T_r\}$ состоит из всех операторных деревьев, ассоциированных с некоторым оптимизируемым запросом. Пусть O_L – алгоритм, который ищет дерево $T_i \in \mathrm{T}$ с наименьшим временем отклика; $O_G(O_H)$ – алгоритм, который ищет дерево $T_i \in \mathrm{T}$ с наименьшим времени отклика в соответствии с функцией *G(H)*. Если полагать, что процедура поиска для O_L,O_G и O_H одинакова, то можно сделать **следующие выводы**:

– алгоритм O_G (O_H) значительно более эффективен алгоритма O_L;

– оптимальное дерево возвращаемое алгоритмом O_G (O_H) почти такое же, как и возвращаемое алгоритмом O_G.

Таким образом, из приведенных теорем следует, что функции G и H, в общем случае, можно считать точными и эффективными функциями оценки времени отклика. Общий принцип проектирования оценочных функций (алгоритмов) состоит в объединении комбинации критического пути и средней работы выполняемой одним процессором.

Заключение

В параллельных распределенных СУБД существует два основных класса оптимизаторов. Первый класс оптимизаторов исследован в настоящей работе. Это, как правило, некоторые сервисы в СУБД, принимающие все поступающие в отдельные узлы SQL запросы. После поступления сервис анализирует этот запрос на его семантическую корректность (наличие запрашиваемых данных из соответствующих объектов СУБД), строит множество планов выполнения запроса (Query Execution Plan – QEP), с использованием оценочных формул строится множество стоимостей всех планов, и среди множества QEP выбирается план, имеющий наименьшую стоимость.

Полученные в работе оценки эффективны в системах кластерного типа, т.е системах с одинаковыми значениями различных коммуникационных и вычислительных параметров. Для анализа оценок времени отклика в распределенных системах, использующих неоднородные вычислительные модули (тактовая частота процессоров-узлов, скорость доступа к данным-архивам на дисках, различные характеристики оперативной памяти в узлах и пр.), описанные модели значительно усложняются необходимостью переопределения времени работы каждого узла и соответствующих подзадач в нем.

В стороне остались проблемы оценки времени отклика на основе статистических методов (второй класс оптимизаторов). Ожидается, что в ближайшие годы распространенными останутся СУБД с данными, хранимыми в традиционно организованной внешней памяти, поэтому традиционные оптимизаторы сохранят свою актуальность.

Автор благодарит доктора технических наук, профессора С.Д. Кузнецова за постановку задачи, полезные и конструктивные обсуждения, а также доктора технических наук В.А. Петрухина и доктора физико-математических наук, профессора Е.М. Лаврищеву за комментарии и предложения.

1. *Griffiths Selinger P., Astrahan M.M., Chamberlain D.D., Lorie R.A. and Price T.G.* Access Path Selection in a Relational Database Management System // *SIGMOD.* – 1979. – P. 23 – 25.
2. *Baru C K, Fecteau G., Goyal A., Hsiao H., Jhingran A., Padmanadhan S., Copeland G. P., Wilson W.G.* DB2 Parallel Edition // *IBM Systems J.* – 1995. – 36, № 2. – P. 293 – 296.
3. *Koval V.N., Savyak V.V., Sergienko I.V.* Tendencies of modern supercomputer systems development // Control Systems and Computers. – 2004. – 6. – P. 31 – 44.
4. DB2 Information Center in the internet. – http://publib.boulder.ibm.com/infocenter/db2help.
5. *Ronstrom M., Thalmann L.* MySQL Clusture Architecture Overview // *MySQL AB.* – 2004. – P. 5 – 6, 10.
6. *Кузнецов С.Д.* Методы оптимизации выполнения запросов в реляционных СУБД. – http://citforum.ru/database/articles/art_26_12.shtml.
7. *Sumit Ganguly, Akshay Goel, Avi Silberschatz*. Efficient and Accurate Cost Models for Parallel Query Optimization // *PODS'96* Montreal Quebec Canada. – 1996. p. 173, 175 – 177.
8. *Shanakar Ramaswamy, Sachin Sapatnekar and Prithviraj Banerjee*. A Convex Programming Approach for Exploiting Data and Functional Parallelism on Distributed Memory Multicomputers // International Conference on Parallel Processing. – 1994. – P. 3 – 4.