Е.В. ПУГИН,
Р.А. СИМАКОВ

Аналитический обзор способов построения процессоров запросов в СУБД

УДК 004.047

Муромский институт (филиал) ФГБОУ ВПО «Владимирский государственный университет имени А.Г. и Н.Г. Столетовых», г.Муром

В статье рассматриваются вопросы построения процессора запросов СУБД. Предлагаются способы по оптимизации времени выполнения запросов.

Системы управления базами данных являются важнейшими инструментами обработки и хранения данных. На сегодняшний день объёмы потоков информации, поступающих из различных источников неуклонно растут. Это могут быть данные с датчиков, снимки звёздного неба с современных телескопов, финансовые данные предприятий и фондовых бирж. Поэтому учёные ведут постоянные работы в областях ускорения обработки этих данных.

Любая СУБД имеет несколько составных частей, таких как хранилище данных, оптимизатор запросов, движок, подсистема управления блокировками, подсистема репликации, сетевая подсистема. Поиски методов оптимизации ведутся в разных компонентах СУБД. Например, учёные проводят исследования по улучшению архитектуры хранилища данных, - замена традиционного хранения кортежей в реляционных СУБД, таких как PostgreSQL [10], Oracle, MS SQL Server, на вертикальные хранилища в распределённых системах; производятся улучшения алгоритмов оптимизации запросов. С другой стороны, в определённых частях СУБД ускорение достигается за счёт разработок и внедрения нового аппаратного обеспечения (более быстрые процессоры, память, сеть). Стоит отметить, что у каждой архитектуры, у каждой оптимизации есть свои сильные и слабые стороны. Поэтому на данный момент всё более распространяется практика написания СУБД, хорошо выполняющих определённый, довольно узкий круг задач. Особенно сильно это проявляется в разработке СУБД для обработки и анализа больших и сверх-

больших объёмов однотипных данных (Vertica, SciDB [9, 15], VoltDB).

В данной статье будет более подробно рассмотрен и проанализирован процессор СУБД, его составляющие, а также возможные способы его построения и оптимизации.

Этапы обработки запроса процессором запросов

Обычно обработка запроса процессором производится в несколько шагов. Вначале запрос к СУБД поступает от клиента. Запрос должен выполняться, как отдельный процесс от прикладного приложения (клиента) для того, чтобы обеспечить защиту данных. В данном случае возникает выбор между моделями архитектуры исполнения запроса [11]:

1. Модель один процесс на одно клиентское соединение (запрос). Преимуществом данной модели является упрощённая реализация по сравнению с серверной архитектурой. Недостатки – более медленное выполнение запросов и сложности коммуникации. При использовании данной модели необходимо использование мастер-процесса, который и будет создавать дочерние процессы, обрабатывающие запросы. Примером СУБД, использующей данный подход является PostgreSQL, Firebird.

2. Серверная архитектура. Один процесс на все запросы. Такая архитектура имеет много преимуществ касающихся скорости выполнения. Это и совместное владение открытых дескрипторов файлов и буферов, оптимизированное переключение задач и упрощённый обмен сообщениями. Данная архитектура имеет преимущества при построении СУБД, нацеленных на ускорение выполнения запросов.

Обмен между сервером и клиентом происходит с использованием сообщений. После отправки запроса, клиент остаётся ждать результатов или сообщения об ошибке.

Первым шагом обработки запроса, представленный на языке запросов (SQL или ином), подвергается лексическому и синтаксическому анализу. При этом вырабатывается его внутреннее представление (план, дерево запроса), отражающее его структуру и содержащее информацию, которая характеризует объекты базы данных, упомянутые в запросе (отношения, поля и константы). Внутреннее

представление запроса используется и преобразуется на следующих стадиях обработки запроса.

На следующем шаге производится поиск правил, хранимых в системных каталогах для применения их к дереву. Одним из примеров применения таких правил является работа с представлениями. Если запрос использует представление, то система подставляет в запрос пользователя запросы, использующие базовые таблицы, как это было указано при определении представлений.

На следующем этапе включается в работу планировщик или оптимизатор [17]. Он принимает переписанный с учётом правил план запроса и создаёт дерево запроса, которое будет использовано исполнителем запросов (query executor). Сначала выполняется так называемая логическая оптимизация. По некоторым правилам производятся различные преобразования, улучшающие начальный план запроса. Среди этих преобразований могут быть эквивалентные преобразования, при которых получается дерево запросов, эквивалентное начальному по смыслу. Также могут выполняться семантические преобразования, при которых новое дерево запроса не является семантически эквивалентным по смыслу, но гарантируется, что результат исполнения этого дерева полностью совпадает с результатом исходного запроса с учётом соблюдения ограничений целостности.

Затем оптимизатор запросов на основе имеющейся информации создаёт набор альтернативных планов выполнения запроса в соответствии с внутренним представлением, полученным на предыдущем этапе. Помимо этого для каждого плана вычисляется предполагаемая стоимость выполнения запроса по этому плану. При оценке используется статистика о состоянии БД, загрузке системы и т.д. Например, если в таблице есть индекс, то возможны два варианта сканирования. Первый включает в себя простое последовательное сканирование, а второй заключается в использовании индекса. Из всех альтернативных планов выбирается тот, который обладает меньшей стоимостью. Он и становится тем планом, который будет передан исполнителю запросов. Пример плана представлен на рис. 1.

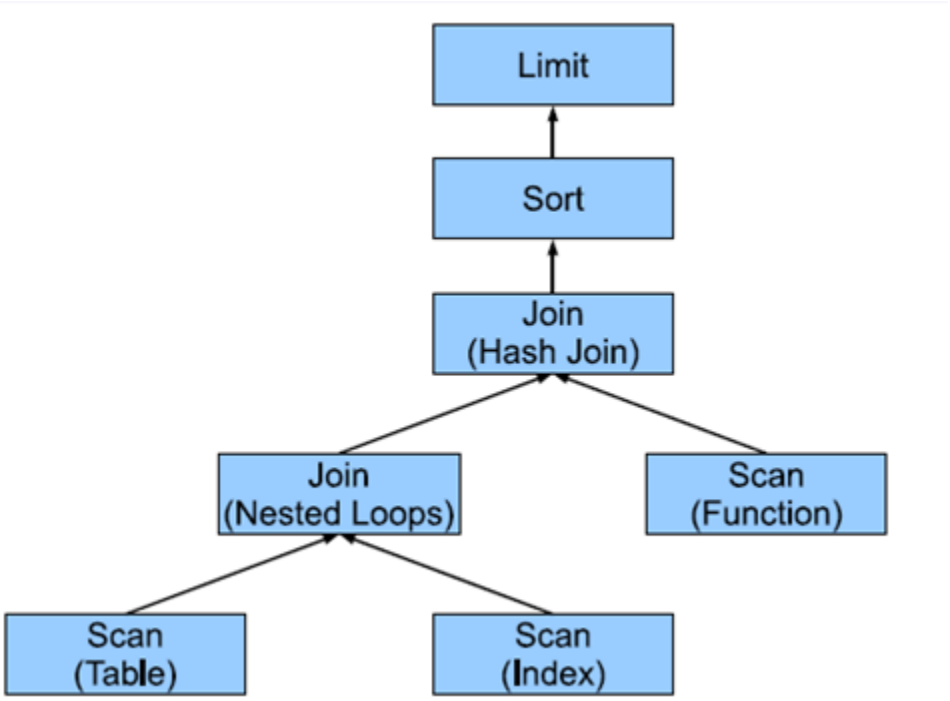

Рис. 1. Пример плана выполнения запроса

Теоретически план запроса должен выглядеть примерно следующим образом (от узлов к корню дерева):

1. Сканирующие операторы, сканирование индексов, функции сканирования, сканирование в подзапросах.
2. Объединения (Joins).
3. Группировки, агрегация.
4. Сортировка.
5. Набор операторов.
6. Проецирование (Projection).

На практике в дереве выполняются различные перестановки и изменения плана, такие как:

1. Перемещение операторов ближе к листьям дерева для уменьшения размера данных. Например, выполнение предикатов и проецирования, как можно раньше.
2. Преобразование подзапросов в объединения.
3. Выбор более подходящих низкоуровневых операторов для получения выгоды от операторов, стоящих выше.

На последнем этапе происходит исполнение запроса. Исполнитель принимает план, созданный оптимизатором и начинает рекурсивно проходить все узлы дерева запроса, чтобы осуществить вы-

борку необходимого набора строк или ячеек. Упрощённая схема процессора запросов показана на рис. 2.

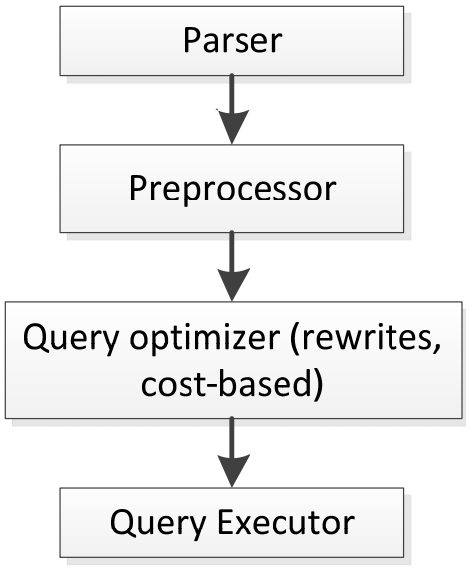

Рис. 2. Схема процессора запросов

Вышесказанное в большей степени относится к традиционным реляционным СУБД, которые уже имеют за собой довольно большую историю и накопленный опыт. Но в настоящее время всё больше набирают популярность NoSQL СУБД, СУБД с вертикальным хранилищем (column-based DBMSs) [2], аналитические распределённые СУБД [9, 16], чья модель данных основывается на многомерных массивах, также развивается аппаратное обеспечение. Всё это приводит к тому, что учёные разрабатывают новые методики по исполнению запросов в таких системах. Ниже рассмотрим некоторые из них более подробно.

Способы построения и оптимизации работы процессоров запросов

При построении, организации и оптимизации процессора запросов необходимо помнить о некоторых базовых принципах, относящихся к работе аппаратного обеспечения и к выполнению запросов:

1. Дисковый ввод/вывод доминирует в стоимости выполнения запроса, поэтому надо избегать частых обращений к дискам. В противоположность этому более интенсивное использование оперативной памяти позволяет ускорить работу системы. С течением времени единица ОП всё больше дешевеет. Стало возможным создание баз данных, полностью хранящих свои данные в ОП. Это приводит к колоссальному ускорению всех операций.

2. Случайный ввод/вывод более дорог, чем последовательные операции, до тех пор, пока операции ввода/вывода не будут закэшированы.

3. Следует уменьшать объём данных, передаваемых между операторами. Как было сказано выше, для этого необходимо применять проекции, группировки и предикаты, как можно раньше (предполагается, что предикаты относительно дёшевы).

Конвейерное исполнение (pipelined execution)

При исполнении запроса может использоваться такой механизм, называемый конвейером (pipeline) [4]. Преимущества конвейерного механизма заключаются в том, что:

1. Оператор не обязательно должен быть полностью выполнен.

2. Конвейерные операторы требуют меньшего объёма памяти для хранения состояния в противоположность материализующим операторам, у которых часто размер результата превосходит объём оперативной памяти, и системе приходится сбрасывать его на диск.

3. Определяет, как поток выполнения команд системой (control flow), так и поток прохождения данных (data flow).

4. Исходя из п. 3, может увеличиваться вероятность попадания данных в кэш процессора, что на порядок увеличивает скорость работы.

5. Операторы просто получают значения ячеек или кортежей на входах и вычисляют результат.

6. Принцип инкапсуляции: каждый оператор не имеет общего представления о выполнении.

Конвейерное исполнение связано с такими понятиями, как итератор и итерационная модель. В такой операторы имеют следующие общие методы:

1. Инициализация. Происходит захват блокировок, инициализация состояния.

2. Получение следующей ячейки или кортежа (GetNext()). 3. Обычно вызывает аналогичную операцию у дочернего оператора. Также возможна поддержка направления операции (получение следующего или предыдущего элемента).

4. Сброс позиции оператора или его состояния, чтобы повторить его выполнение сначала.

5. Запись текущего состояния оператора или его позиции.

6. Восстановление предварительно записанного состояния.

7. Завершение. Происходит освобождение ресурсов.

Bitmap Scans

В СУБД с вертикальным хранилищем, а также в СУБД, которые работают с многомерными массивами удобно в качестве индексов использовать битовые карты [8]. Основная идея заключается в том, чтобы отделить сканирование индекса от сканирования данных. В реляционных СУБД для каждого индекса требуемой таблицы следует создать битовую карту. Для этого сканируется индекс, чтобы найти удовлетворяющие нас записи. Затем эти записи отображаются в битовую карту, хранящуюся в оперативной памяти. Используется 1 бит на каждую запись, если есть пустые записи. В противном случае – 1 бит на каждую страницу. Для многомерных СУБД битовая карта показывает наличие или отсутствие ячеек (empty), пустые ячейки (null), а в некоторых случаях и всё вместе. Также преимущества битовых карт заключаются в том, что логические операции в данном случае выполняются не над данными, а над битовыми картами, что во много раз ускоряет выполнение этих операций, т.к. не требует чтение самих данных, а только их битовых карт. Преимущества битовых карт:

1. После нахождения результирующих битовых карт организуется последовательное, а не соответствующее индексу (аналогичное случайному), сканирование данных, что также сказывается на производительности.

2. Битовые карты позволяют создавать комбинации нескольких индексов для одной таблицы [12].

3. Более гибкие, чем индексы по нескольким полям.

Параллельная обработка и распределённые СУБД

На текущий момент аппаратное обеспечение компьютеров сделало большой скачок в производительности и объёму памяти на единицу стоимости по сравнению с 1990-2000-ми годами. Скорости передачи в сетях выросли до 10-40 Гбит/с, процессоры содержат по несколько физических ядер, стоимость оперативной и дисковой памяти значительно упала. Это позволяет организовывать огромные кластерные системы, способные обрабатывать массивы данных объёмом до нескольких Петабайт. Естественно все эти особенности

должны учитываться в архитектуре СУБД и при выполнении запросов.

Процессор запросов должен взаимодействовать с менеджером ресурсов (управление параллельной обработкой на локальном узле), а также сетевой подсистемой, чтобы организовывать выполнение в параллельном режиме на локальном и на других узлах сети. В таких системах оптимизатору необходимо уметь работать с учётом распределённой структуры и обработки.

Улучшение процессоров происходит не только в направлении роста тактовых частот и увеличения числа ядер. Увеличивается объём кэша (до 1-2 Мб L2), а также появляются новые векторные команды обработки данных.

Попадания данных и инструкций в кэш является одной из наиболее сильно влияющих на производительность причин. Необходимо учитывать это, как и при исполнении запросов (сокращение числа вызовов функций, сбрасывающих конвейер процессора, конвейерная обработка порций данных внутри СУБД), так и при организации структуры данных (размеры чанков и т.д.).

Векторные инструкции, такие как сложение, умножение, вычитание, совмещённое умножение-сложение (FMA) позволяют обрабатывать процессору по несколько ячеек памяти за 1 такт. Эту особенность следует учитывать при проектировании исполнителей запросов, т.е. перед операциями производить необходимую подготовку данных. Скорость обработки соответственно увеличивает во столько раз, во сколько раз больше процессор обрабатывает за 1 такт с помощью векторной инструкции. Например, в микроархитектуре Intel MIC обеспечивается обработка до 512 бит (8 ячеек по 8 байт) данных с плавающей точкой за 1 такт, что составляет 8-кратный прирост производительности. Векторизация вычислений на программном уровне связана со сжатием обрабатываемых данных.

Следует отметить, что из-за удешевления единицы оперативной памяти, становятся доступными пока ещё не такие большие in-memory базы данных. Это БД, данные которых всегда находятся в ОП компьютера и не подлежат сбросам на диск. Производительность обработки такой информации очень велика, а число операций дискового ввода/вывода может ограничиваться только начальной загрузкой данных. В данном случае процессор запросов должен

быть изменён с учётом этих факторов. А главным критерием, влияющим на дальнейшее ускорение работы будет оптимизация числа попаданий данных в кэш процессора.

Технологии сжатия данных

В традиционных СУБД известно, что сжатие данных значительно увеличивает производительность [3, 5]. Оно уменьшает размер данных и увеличивает производительность операций ввода/вывода путём уменьшения затрат времени на поиск, так как данные хранятся ближе друг к другу. Сжатие данных снижает время передачи данных и увеличивает вероятность попадания в буфер. Следует отметить, что СУБД с вертикальным хранилищем получают более значительные преимущества по сравнению с реляционными СУБД [13]. Это происходит потому, что в реляционных СУБД организовано горизонтальное хранение данных, т.е. в виде записей. Это приводит к тому, что сжатие выполняется на сильно разнородных данных, что снижает его эффективность. В СУБД с вертикальным хранилищем данные хранятся по атрибутам, что делает сжатие однородных данных очень эффективным.

Обработка сжатых данных может выполняться более эффективно и даже в векторном режиме. К примеру, если число 42 повторяется в атрибуте 1000 раз, то, вычисляя агрегат SUM, мы просто можем умножить число повторений на само число, вместо 1000 операций сложения.

Существует несколько видов сжатия данных:

1. Null suppression – подавление пустых и нулевых ячеек. Существует множество вариаций этого метода, но основная идея этого метода заключается в том, что последовательные нулевые и пустые ячейки удаляются и заменяются описанием, как много их было, и где они находились.

2. Dictionary encoding – кодирование по словарю. Является наиболее распространённой схемой кодирования в СУБД на сегодняшний день. Проявляется в виде замены часто повторяющихся шаблонов на более маленькие коды для них. Использование данного метода также возможно и в многомерных СУБД.

3. Run-length encoding (RLE) – кодирование длин серий. Простой алгоритм, который оперирует сериями данных, то есть последовательностями, в которых один и тот же символ встречается несколь-

ко раз подряд. Каждая такая последовательность заменяется компактным представлением (значение, начальная позиция, длина). Данный способ очень хорошо подходит для многомерных СУБД и СУБД с вертикальным хранилищем данных. В реляционных СУБД в основном используется только для хранения длинных строковых атрибутов, которые имеют много пробелов или повторяющихся символов.

4. Bit-vector encoding – кодирование в битовых векторах. Данное кодирование наиболее полезно тогда, когда атрибуты имеют ограниченное число возможных значений (например, хранение в атрибуте некоторых флагов). Суть метода заключаются в следующем: битовая строка ассоциируется с каждым значением так, что если это значение встречено, то в соответствующую позицию битовой строки записывается 1, в противном случае 0. Пример: имеются возможные значения – 1, 2, 3 и набор значений в атрибуте:

1 1 3 2 2 3 1

Тогда эта последовательность значений с использованием данного метода будет представлена, как:

Битовая строка для значения 1: 1 1 0 0 0 0 1
Битовая строка для значения 2: 0 0 0 1 1 0 0
Битовая строка для значения 3: 0 0 1 0 0 1 0

Существуют также другие более сложные алгоритмы кодирования. Например, алгоритм Лемпеля-Зива для сжатия файлов [7] показал свою пригодность к использованию, а алгоритмы кодирования Хаффмана [6] и арифметического кодирования непригодны из-за высокой стоимости распаковки.

Само по себе сжатие данных позволяет увеличить производительность системы, но интеграция подсистем сжатия и обработки данных [1] даёт возможность дополнительно ускорить процесс исполнения запросов, то есть проведение вычислений, как это показано в наглядном примере для метода RLE. В этом случае также необходимо производить соответствующие изменения в схеме процессора и оптимизаторе запросов, которые должны учитывать эти особенности.

Push и pull модели процессоров

В большинстве СУБД используется pull (от англ. тянуть) модель процессора запросов. Это означает, что после получения оконча-

тельного плана от оптимизатора запросов, происходит последовательное выполнение операторов по дереву от узлов к корню. Инициатором обработки в этом случае является процессор запросов. При таком подходе данные запрашиваются системой и «протянутся» через цепочки операторов. Часто такая обработка организуется с использованием итераторов и принципа конвейерной обработки. Процессор запросов в данном случае имеет точный детерминированный план выполнения.

Второй моделью является push (от англ. толкать) процессор запросов (push executor). Обычно такая модель используется в peer-to-peer и сетевых процессорах запросов [14]. Они обрабатывают данные, сильно распределённые по сети. Очень часто данные для обработки могут находиться даже во внешних хранилищах, не относящихся к СУБД. При таком подходе используется гибкое планирование (flexible scheduling) выполнения запроса. Это позволяет центральному процессору обрабатывать различные части плана, когда он сталкивается с задержкой при ожидании определённого источника данных. Многие системы для обработки распределённых данных часто внедряют “push” операторы, вместо традиционных моделей, основанных на итераторах (pull). Примером может служить оператор – pipelined hash join. Проблемой при использовании данной модели может стать включение в запрос «блокирующих» операторов и множественных операторов объединения, т.е. таких операторов, которые требуют на вход сразу все данные. Чтобы снизить влияние таких операторов на производительность, используются специальные методы (фильтры Блума, хеш-фильтры, semijoins, Bloomjoins), позволяющие за счёт передачи дополнительной информации внутри запроса сократить время исполнения.

Сравнение характеристик процессоров запросов в различных СУБД

Для сравнения различных способов построения процессора запросов составим сводную таблицу его характеристик в различных СУБД. Так как он строится с учётом архитектуры всей системы в целом и зависит от решений, принятых при проектировании других подсистем, модели данных и других требований к СУБД, в качестве признаков будем использовать некоторые свойства конкретной СУБД. В таблице рассмотрим следующие характеристики:

1. Тип СУБД (реляционная - Р, многомерная - М, хранилище пар ключ-значение - Х).

2. Организация данных (горизонтальная - Г, вертикальная - В).

3. Распределённость.

4. Многопоточное выполнение запроса.

5. Push/pull модель или смешанная (mixed).

6. Конвейерная обработка и использование итераторов.

7. Архитектура распределённой обработки (shared memory - SM, shared disk – SD, shared nothing – SN, shared everything - SE) [16].

8. Сжатие данных.

Таблица 1

Сводная таблица характеристик процессоров запросов в различных СУБД

СУБД	Характеристики							
	1	2	3	4	5	6	7	8
PostgreSQL	Р	Г	-	-	pull	+	SM	?
Oracle database	Р	Г	+	+	pull	?	?	?
Firebird	Р	Г	-	-	pull	?	?	?
MySQL	Р	Г	+	-	mixed	?	SN	?
MS SQL Server	Р	Г	+	+	pull	?	?	?
Vertica	Р	В	+	+	pull	+	SN	+
SciDB	М	В	+	+	pull	+	SN	+
MonetDB	Р	В	?	+	pull	+	?	+
Sybase IQ	Р	В	+	+	pull	?	SE	+

Заключение

В данной статье был рассмотрен один из компонентов любой СУБД – процессор запросов, описаны основные его составные части, а также предложены методы и алгоритмы его разработки и улучшения. Изложены возможные оптимизации этих методов с учётом современных тенденций в области аппаратного обеспечения и потребностей пользователей. Показаны преимущества и недостатки отдельных способов. Построение процессора во многом зависит от остальной архитектуры СУБД и от способа организации данных.

Можно говорить о том, что потребности пользователей смещаются в сторону сверхбольших хранилищ вертикальных данных и уходят от традиционных реляционных СУБД. Поэтому многие из предложенных методов наиболее эффективно показывают себя именно в таких системах. Оцениваемый показатель прироста производительности может достигать двух порядков.

В качестве перспективных направлений по дальнейшему увеличению производительности можно выделить следующие: программная векторизация вычислений и адаптация аппаратной векторизации, эффективное сжатие и обработка сжатых данных, а также повышение вероятности попадания данных в кэш процессора.

Литература

1. Daniel J. Abadi, Samuel R. Madden, and Miguel Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In Proceedings of SIGMOD, 2006.

2. Daniel J. Abadi. Query Execution in Column-Oriented Database Systems. MIT PhD Dissertation, February, 2008.

3. Balakrishna R. Iyer and David Wilhite. Data compression support in databases. In VLDB ’94, pages 695–704, 1994.

4. Peter Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-pipelining query execution. In CIDR, 2005.

5. Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing relations and indexes. In ICDE ’98, pages 370–379, 1998.

6. G.Graefe and L.Shapiro. Data compression and database performance. In ACM/IEEE-CS Symp. On Applied Computing pages 22 -27, April 1991.

D. Huffman. A method for the construction of minimum-redundancy codes. Proc. IRE, 40(9):1098-1101, September 1952.

7. Jacob Ziv and Abraham Lempel; Compression of Individual Sequences Via Variable-Rate Coding, IEEE Transactions on Information Theory, September 1978.

8. Theodore Johnson. Performance measurements of compressed bitmap indices. In VLDB, pages 278–289, 1999.

9. Overview of SCIDB: Large scale array storage, processing and analysis. Rogers J., Zdonik S., Simakov R., Velikhov P., Smirnov A., Knizhnik K., Soroush E., Balazinska M., DeWitt D., Heath B., Maier D., Madden S., Stonebraker M., Patel J., Brown P.G. Proceedings of the ACM SIGMOD International Conference on Management of Data 2010 International Conference on Management of Data, SIGMOD '10. Сер. "Proceedings of the 2010 International Conference on Management of Data, SIGMOD '10" sponsors: ACM SIGMOD. Indianapolis, IN, 2010. С. 963-968.

10. M. Stonebraker and L. Rowe, " The design of POSTGRES ", Proc. ACM-SIGMOD Conference on Management of Data, May 1986.

11. Stonebraker, M., ‘‘Operating System Support for Database Management,’’ CACM, July 1981.

12. Patrick O’Neil and Goetz Graefe. Multi-table joins through bitmapped join indices. SIGMOD Rec., 24(3):8–11, 1995.

13. Terry Welch, "A Technique for High-Performance Data Compression", IEEE Computer, June 1984, p. 8–19.

14. Z. G. Ives and N. E. Taylor, “Sideways information passing for push-style query processing,” Tech. Rep. MS-CIS-07-14, University of Pennsylvania, 2007.

15. Пугин Е.В., Симаков Р.А. Распараллеливание математических вычислений на примере операторов линейной алгебры. / Алгоритмы, методы и системы обработки данных. 2011. № 17. С. 12-12.

16. Симаков Р.А., Смяткин М.А. Особенности архитектуры распределенной массивно-реляционной СУБД. / Алгоритмы, методы и системы обработки данных. 2011. № 18. С. 9-9.

17. Смирнов А.В., Пугин Е.В. Разработка модели оптимизатора запросов. / Алгоритмы, методы и системы обработки данных. 2010. № 15. С. 161-169.

ПУГИН Е.В.

E-MAIL: EGOR.PUGIN@GMAIL.COM