Лекции по курсу "Параллельные системы баз данных"

Соколинский Леонид Борисович, Цымблер Михаил Леонидович

Оглавление

1. Основы технологии построения параллельных систем баз данных

Данный раздел содержит минимальный теоретический материал, необходимый для осознанного выполнения лабораторного практикума по курсу «Параллельные системы баз данных».

1.1 Учебная база данных «Поставки»

В примерах мы будем использовать учебную базу данных «Поставки», изображенную на Рис. 1. База данных «Поставки» состоит из трех таблиц: П, Д и ПД. Таблица Д содержит информацию о деталях, необходимых для производства некоторых устройств, собираемых на нашем предприятии. Таблица П содержит информацию о поставщиках, поставляющих эти детали. Таблица ПД содержит записи о том, какой поставщик какие детали поставляет. Символом * помечены первичные ключи. Символом # помечены внешние ключи.

Рис. 1. Учебная база данных «Поставки»

1.2 Формирование последовательного плана

Рис. 2 демонстрирует формирование *последовательного (не параллельного)* плана запроса. Пользователь формулирует запрос на языке SQL. Компилятор преобразует его в выражение реляционной алгебры, представляемого в виде древовидной структуры, называемой *планом запроса*.

Для обозначения реляционных операций здесь используется следующая нотация [8]:

- Проекция: $\pi_i(R) = \{(r_i) \mid (r_1,..., r_i,... r_n) \in R \ \& \ \}$;
- Выборка по условию P: $\sigma_P(R) = \{r \mid r \in R \ \& \ P(r)\}$;
- Естественное соединение: $\triangleright\triangleleft$.

В дереве, представляющем план запроса, листьями являются отношения (таблицы), а в качестве узлов фигурируют реляционные операции. В качестве сыновей узла выступают операнды реляционной операции, представляющей данный узел. В данном примере план запроса соответствует следующему выражению реляционной алгебры:

$$\pi_{\text{Имя\_П}}(\sigma_{\text{Город = 'Москва'}}(\text{П}) \triangleright\triangleleft (\text{ПД} \triangleright\triangleleft \sigma_{\text{Цвет = 'Красный'}}(\text{Д})))$$

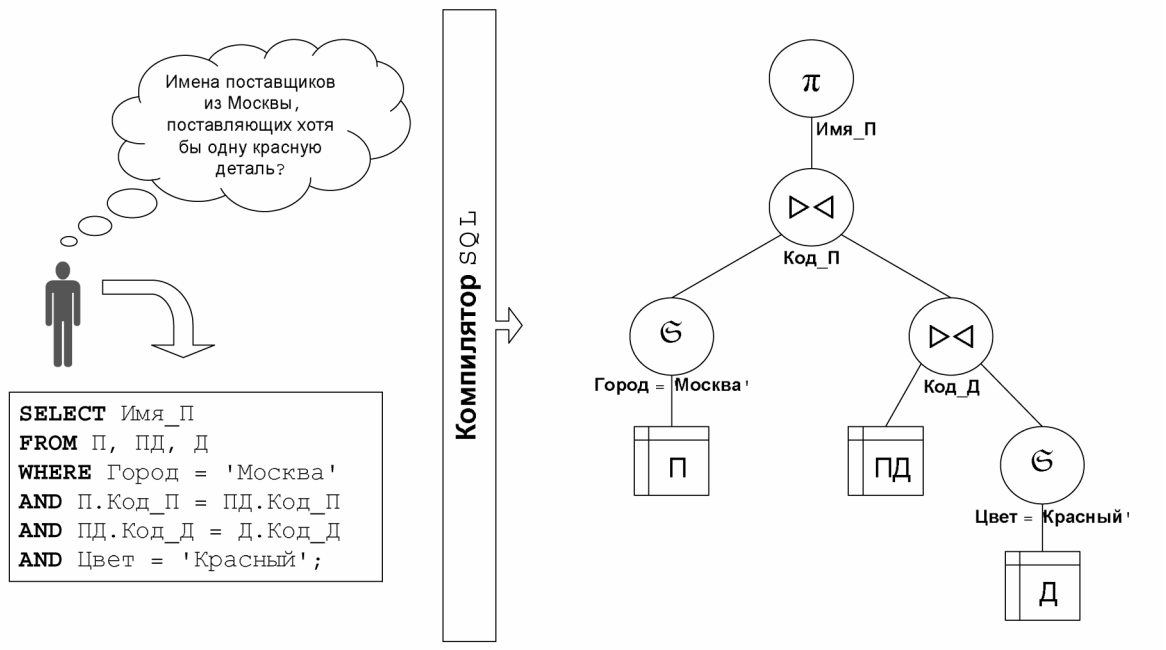

Рис. 2. Формирование последовательного плана

Реализация исполнителя последовательных планов базируется на следующих трех базовых парадигмах:

- синхронный конвейер;
- итераторная модель;
- скобочный шаблон.

Следует сразу отметить, что данные методы являются ортогональными по отношению к методам параллелизации запросов, описываемых ниже. Это означает, что синхронный конвейер, итераторная модель и скобочный шаблон могут быть использованы с равным успехом и при реализации параллельных СУБД.

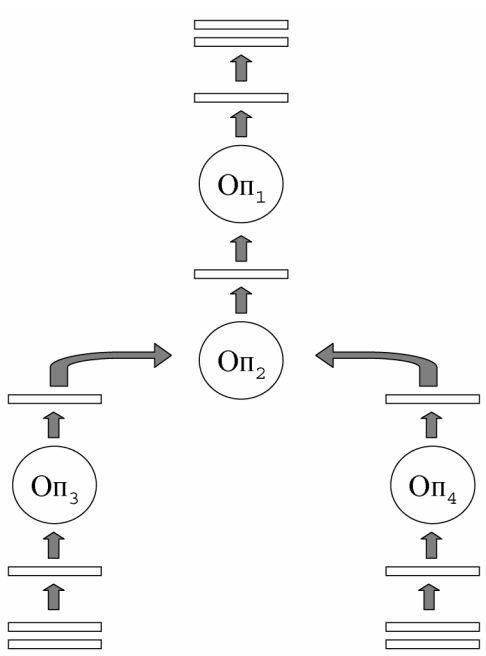

Рис. 3. Синхронный конвейер

1.3 Синхронный конвейер

Выполнение запроса в базах данных обычно связано с обработкой очень больших отношений. Под словом «очень» мы подразумеваем тот факт, что отношение базы данных не помещается целиком в оперативную память. В этих условиях мы вправе ожидать, что промежуточные отношения, возникающие при выполнении плана запроса также могут быть *очень* большими. Стандартным приемом, применяемым в СУБД для решения этой проблемы, является организация между операциями в дереве плана запроса так называемого *синхронного конвейера* для передачи кортежей промежуточных отношений. Суть данного метода состоит в том, что, как только операция получает очередной кортеж своего результирующего отношения, она немедленно передает его *по конвейеру* выше стоящей операции для обработки (см. Рис. 3).

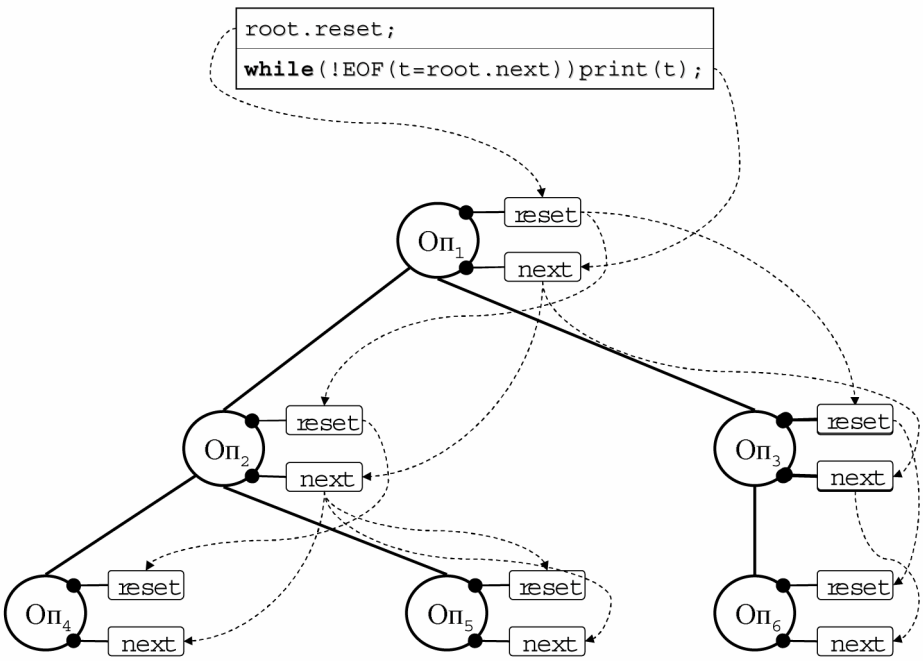

Рис. 4. Алгоритм выполнения плана запроса на базе итераторной модели

Синхронный конвейер допускает простую и эффективную реализацию. Основным недостатком синхронного конвейера является то, что задержка на любом участке конвейера приводит к остановке всего конвейера в целом.

Отметим, что в некоторых параллельных СУБД используется *асинхронный конвейер*, предполагающий организацию между операцией-производителем и операцией-потребителем *склада* (специального буфера) для хранения некоторого количества кортежей промежуточного отношения. Это обеспечивает некоторую независимость участников конвейера. Асинхронный конвейер легко превратить в синхронный, установив размер склада равным одному кортежу.

1.4 Итераторная модель

Итераторная модель является общепринятым методом, используемым в СУБД для эффективной реализации синхронного конвейера. В соответствие с итераторной моделью с каждым узлом дерева плана запроса связывается специальная структура управления, называемая итератором. Интерфейс итератора представлен двумя стандартными операциями с предопределенной семантикой:

- reset – установка итератора в состояние «перед первым кортежем»;
- next – выдать очередной кортеж результирующего отношения.

Алгоритм выполнения плана запроса на базе итераторной модели изображен на Рис. 4. На первом шаге выполняется метод **`reset`** применительно к корневому узлу. Затем в цикле выполняется метод **`next`** для корневого узла. Он каждый раз возвращает указатель на очередной кортеж результирующего отношения. В приведенном примере эти кортежи просто выводятся на экран. Цикл завершается, когда метод **`next`** выдает указатель на специальный кортеж, обозначающий конец файла – EOF (End Of File).

Методы **`reset`** и **`next`** родителя прямо или косвенно могут вызывать соответствующие методы дочерних узлов. Эти вызовы изображены на слайде пунктирными стрелками. Реализация итератора базируется на *скобочном шаблоне*, обсуждаемом в следующем разделе.

```
void node.reset{
   node.bof = 1;
   node.eof = 0;
   node.leftSon.reset;
   if(node.rightSon != null)
      node.rightSon.reset;
};
```

Рис. 5. Схема реализации метода `reset`

```
buffer node.next{
   node.bof = 0;
   node.op(node);
   if(node.buffer==EOF)
      node.eof = 1;
   return node.buffer;
};
```

Рис. 6. Схема реализации метода `next`

1.5 *Скобочный шаблон*

Для унифицированного представления узлов дерева запроса используется класс «скобочный шаблон». Схематично структура скобочного шаблона изображена на Рис. 7. В качестве основных методов здесь фигурируют **`reset`** и **`next`**, реализующие итератор. Основными атрибутами скобочного шаблона являются

- выходной буфер, в который помещается очередной кортеж результата;
- КОП – код реляционной операции, реализуемой данным узлом;
- указатель на скобочный шаблон левого сына;
- указатель на скобочный шаблон правого сына («пусто» для унарных операций).

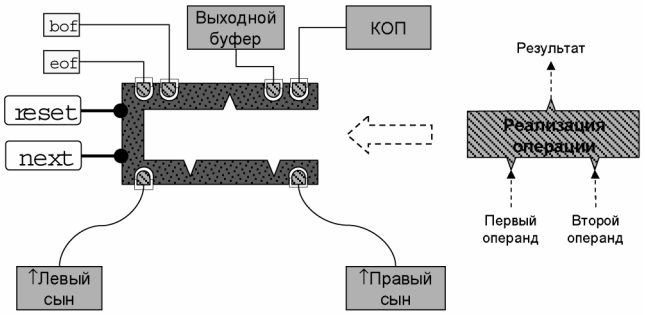

Рис. 7. Скобочный шаблон

Сам по себе скобочный шаблон не содержит конкретной реализации реляционной операции. Однако, после оптимизации запроса СУБД «вставляет» в каждый скобочный шаблон ту или иную реализацию соответствующей реляционной операции. Например, для операции соединения мы можем выбирать «соединение вложенными циклами», «соединение слиянием», соединение хешированием» и др. Связь скобочного шаблона с конкретной реализацией операции осуществляется путем добавления еще одного специального атрибута в скобочный шаблон – «указатель на функцию реализации операции». В качестве параметра данной функции должен передаваться указатель на объемлющий скобочный шаблон.

Схема реализации методов **`reset`** и **`next`** для узлов, не являющихся листьями, изображена на Рис. 5 и Рис. 6. Для записи алгоритмов используется Си-подобный псевдокод. *Реализация метода* ***`reset`*** состоит в выполнении **`reset`** для левого и правого (если не пуст) сыновей. Для унарных операций правый сын всегда содержит пустую ссылку. *Реализация метода* ***`next`*** состоит в выполнении реализации операции (РОП), «вставленной» в данный скобочный шаблон. РОП должна вычислить очередной кортеж результата и поместить его в выходной буфер скобочного шаблона. При этом РОП может использовать методы **`reset`** и **`next`** применительно к сыновьям шаблона-хозяина.

```
/* Соединение вложенными циклами */
void* NL_join(node){
#define L node.leftSon.buffer   // левый операнд
#define R node.rightSon.buffer  // правый операнд
   if(node.leftSon.bof)   // Если "начало файла" у левого сына,
      node.leftSon.next;  //    выполнить next для левого сына.
   if(node.leftSon.eof){  // Если левый сын вернул eof,
      node.buffer=EOF;    //    записать в выходной буфер eof и
      return node.buffer; //    закончить работу.
   };
   do{
      node.rightSon.next;        // Выполнить next для правого сына.
      if(node.rightSon.eof){     // Если правый сын вернул eof,
         node.rightSon.reset;    //    выполнить reset для правого сына и
         node.rightSon.next;     //    повторно выполнить next.
         if(node.rightSon.eof){ // Если опять получили eof,
            node.buffer=EOF;    //       записать в выходной буфер eof и
            return node.buffer; //       закончить работу.
         };
         node.leftSon.next;      // Выполнить next для левого сына.
         if(node.leftSon.eof){  // Если левый сын вернул eof,
            node.buffer=EOF;    //    записать в выходной буфер eof и
            return node.buffer; //    закончить работу.
         };
      };
   }while(!MATCH(L,R));   // пока не совпали атрибуты соединения.
   node.buffer=L°R;   // Поместить в выходной буфер конкатенацию L и R.
   return node.buffer;
};
```

Рис. 8. Реализация соединения вложенными циклами

На Рис. 8 приведен пример РОП для операции соединения, использующий алгоритм соединения вложенными циклами. Следует отметить, что алгоритм соединения вложенными циклами допускает обобщение на случай *фрагментного параллелизма* (см. раздел 1.6) только в том случае, когда отношение, сканируемое во внутреннем цикле, фрагментировано по атрибуту соединения. В общем случае для параллельного соединения обычно используется *Grace алгоритм* [10].

1.6 Фрагментный параллелизм

Основной формой параллельной обработки запросов является *фрагментный параллелизм* (см. Рис. 9). Каждое отношение делится на части, называемые *фрагментами*. Фрагменты отношения распределяются по различным процессорным узлам многопроцессорной системы. Способ фрагментации определяется *функцией фрагментации* ψ, которая для каждого кортежа отношения вычисляет номер процессорного узла, на котором должен быть размещен этот кортеж. На Рис. 9 изображена фрагментация отношения П по атрибуту Код_П на основе *метода диапазонов*. В данном случае функция фрагментации имеет вид: ψ(п) = п.Код_П div 10. Здесь div – операция деления нацело.

В простейшем случае запрос параллельно выполняется на всех процессорных узлах. Полученные фрагменты *сливаются* в результирующее отношение. На практике, однако, не удается избежать пересылки кортежей между процессорами во время выполнения запроса.

Рассмотрим пример, изображенный на Рис. 10. Пусть отношение П фрагментировано по атрибуту Код_П, а отношение ПД – по атрибуту Код_Д. При выполнении операции соединения мы должны динамически перераспределять кортежи отношения ПД, так как оно фрагментировано *не по атрибуту соединения*. Способ перераспределения определяется функцией распределения δ, которая для каждого кортежа отношения вычисляет номер процессорного узла, на котором должен быть обработан этот кортеж.

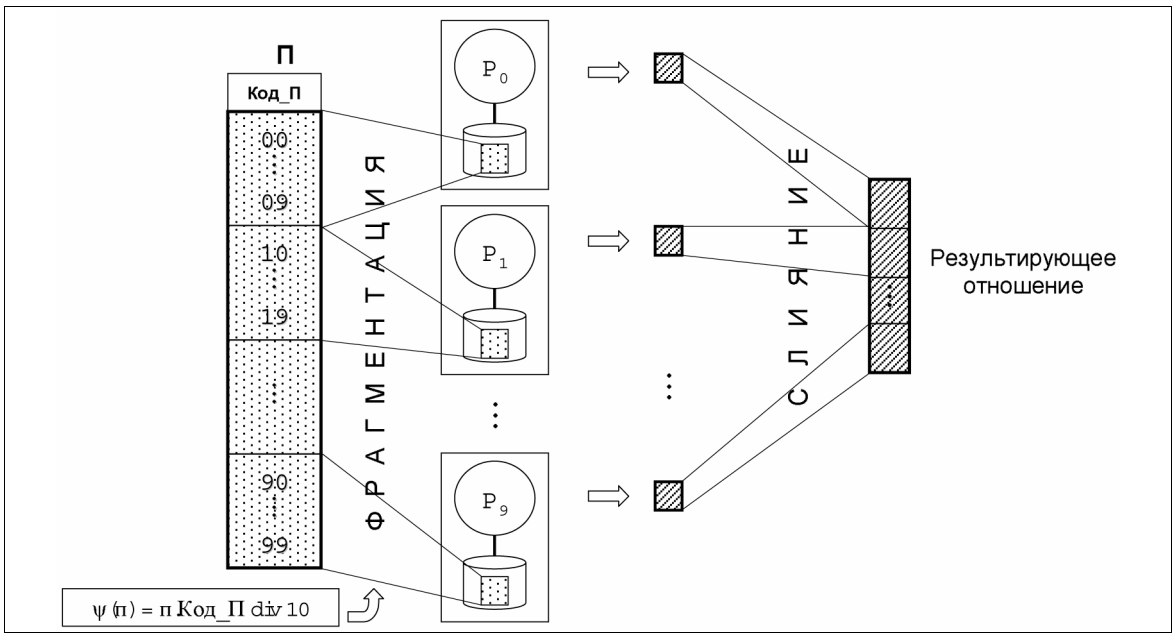

Рис. 9. Фрагментный параллелизм

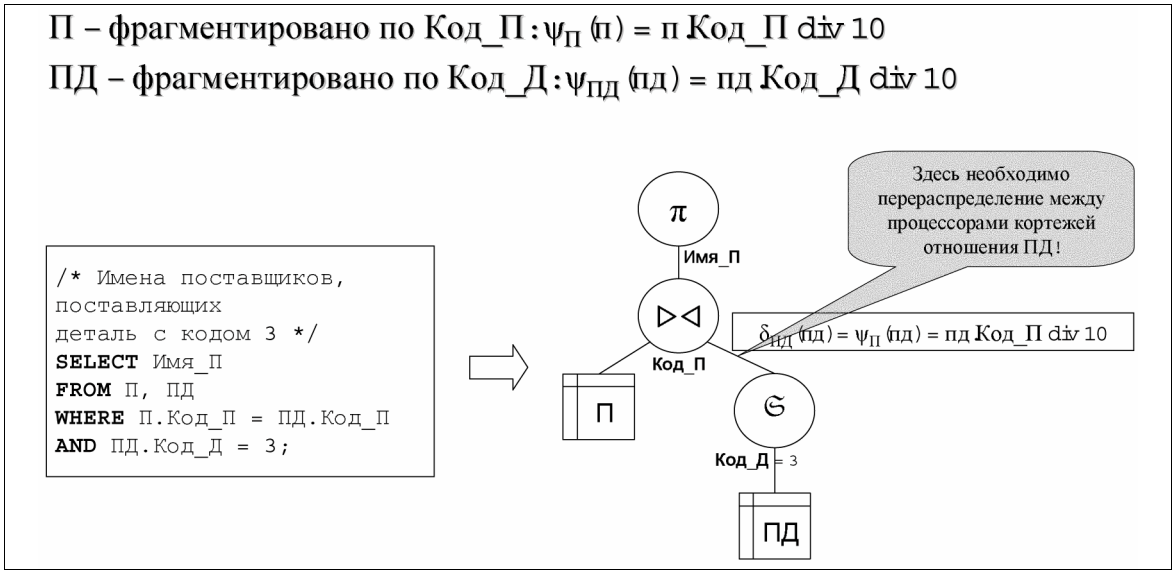

Рис. 10. Необходимость межпроцессорных обменов

В данном примере в качестве функции распределения для ПД следует взять функцию фрагментации отношения П:

$\delta_{ПД}$(пд) = $\psi_{П}$(пд) = пд.Код_П div 10.

Если оба аргумента соединения фрагментированы не по атрибуту соединения, нам придется перераспределять между процессорами оба входных отношения. При этом в качестве функции распределения для обоих отношений мы можем взять любую, но одну и ту же, функцию распределения по атрибуту соединения, которая отправляет кортежи с одинаковыми значениями атрибута соединения на один и тот же процессорный узел.

Для организации межпроцессорных обменов в соответствующие места дерева плана запроса вставляется специальный оператор **exchange**, обсуждаемый в следующем разделе.

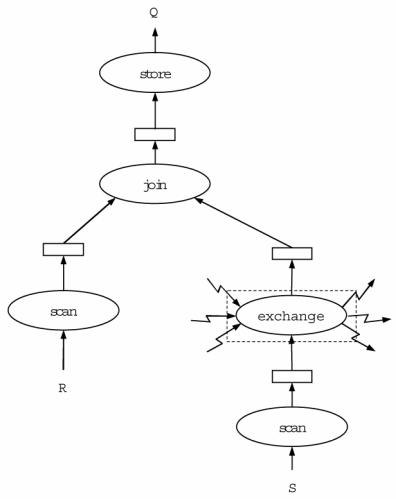

Рис. 11. Дерево плана запроса Q = R ◁▷ S

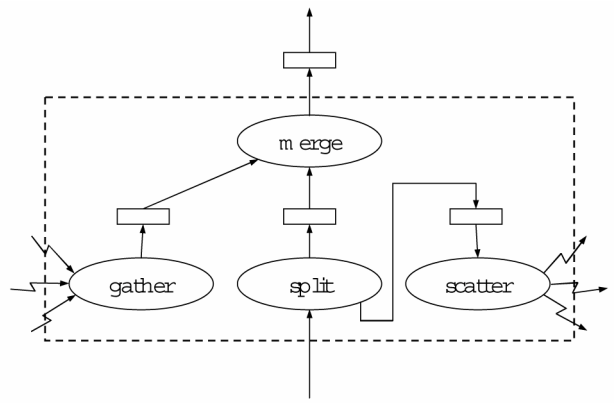

Рис. 12. Структура оператора exchange

1.7 Оператор exchange

Для организации межпроцессорных обменов используется специальный оператор **exchange**. Оператор **exchange** реализуется на основе использования стандартного скобочного шаблона и может быть добавлен в качестве узла в любое место дерева запроса.

Оператор **exchange** имеет два специальных параметра, определяемых пользователем: номер *порта обмена* и указатель на *функцию распределения*. Функция распределения для каждого кортежа вычисляет логический номер процессорного узла, на котором данный кортеж должен быть обработан. Параметр "порт обмена" позволяет включать в дерево запроса произвольное количество операторов **exchange** (для каждого оператора указывается свой уникальный порт обмена). Пример использования оператора обмена **exchange** для распараллеливания запроса приведен на Рис. 11. Здесь изображен физический план выполнения запроса, реализующего соединение двух отношений *R* и *Q* по некоторому общему атрибуту. Мы предполагаем, что отношение *R* фрагментировано по атрибуту соединения с помощью некоторой функции ψ_R, а отношение *Q* фрагментировано по некоторому другому атрибуту, не являющемуся атрибутом соединения. В данном контексте необходимо вставить в дерево запроса между оператором чтения **scan Q** и оператором соединения **join** один оператор обмена **exchange**. В качестве функции распределения оператора **exchange** указывается функция ψ_R, а в качестве номера порта обмена - любой свободный на данный момент номер.

Структура оператора обмена ***exchange*** изображена на Рис. 12. Оператор **exchange** является составным оператором и включает в себя четыре оператора: **gather**, **scatter**, **split** и **merge**. Все указанные операторы реализуются на базе стандартного скобочного шаблона.

Оператор **split** - это бинарный оператор, который осуществляет разбиение кортежей, поступающих из входного потока (ассоциируется с левым сыном), на две группы: *свои* и *чужие*. Свои кортежи - это кортежи, которые должны быть обработаны на данном процессорном узле. Эти кортежи направляются в выходной буфер оператора **split**. Чужие кортежи, то есть кортежи, которые должны быть обработаны на процессорных узлах, отличных от данного, помещаются оператором **split** в выходной буфер правого сына, в качестве которого фигурирует оператор **scatter**. Здесь выходной буфер оператора **split** играет роль входного потока данных.

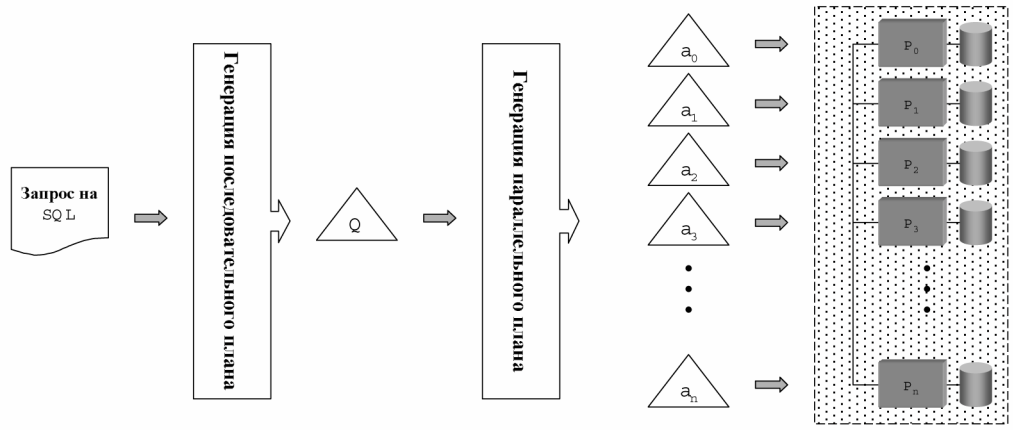

Рис. 13. Обработка запроса в параллельной СУБД

Нульарный оператор **scatter** извлекает кортежи из своего выходного буфера и пересылает их на соответствующие процессорные узлы, используя заданный номер порта. Запись кортежа в порт может быть завершена только после того, как реципиент выполнит операцию чтения из данного порта.

Нульарный оператор **gather** выполняет перманентное чтение кортежей из указанного порта со всех процессорных узлов, отличных от данного. Считанные кортежи помещаются в выходной буфер оператора **gather**.

Оператор **merge** определяется как бинарный оператор, который забирает кортежи из выходных буферов своих сыновей и помещает их в собственный выходной буфер.

1.8 Обработка запроса в параллельной СУБД

Общая схема обработки запроса в параллельной СУБД изображена на Рис. 13. В соответствие с этой схемой запрос на языке SQL преобразуется в некоторый *последовательный план*. Данный последовательный план преобразуется в *параллельный план*, представляющий собой совокупность *n* идентичных параллельных *агентов*. Здесь *n* обозначает количество процессорных узлов. Это достигается путем вставки оператора обмена **exchange** в соответствующие места дерева плана запроса. На завершающем этапе агенты рассылаются на соответствующие процессорные узлы, где *интерпретируются* исполнителем запросов. Результаты выполнения агентов объединяются *корневым* оператором **exchange** на нулевом процессорном модуле.

Поясним цикл обработки запроса в системе Омега на следующем примере (Рис. 14). Пусть необходимо вычислить $\mathbf{Q} = \mathbf{R} \triangleright\triangleleft \mathbf{S}$ – соединение двух отношений **R** и **S** по некоторому общему атрибуту *Y*. Предположим, что отношение *R* фрагментировано произвольным образом, а отношение *S* – по атрибуту соединения *Y*. Предположим, что наша система имеет 4 процессорных узла. В этом случае последовательный план *Q* преобразуется в четырех параллельных агентов так, как это показано на Рис. 14. Все агенты идентичны, за исключением индексов исходных фрагментов отношений.

Оператор $\mathbf{exchange}_{\varepsilon_1}$ имеет функцию распределения $f(r) = 0$. Это означает, что все кортежи, полученные в результате выполнения операции **join**, должны быть отправлены на нулевой процессорный узел (то есть для агентов A_1, A_2, A_3 выходной поток оператора $\mathbf{exchange}_{\varepsilon_1}$ будет всегда пуст).

В качестве функции распределения оператора $\mathbf{exchange}_{\varepsilon_2}$ используется функция фрагментации отношения *S*.

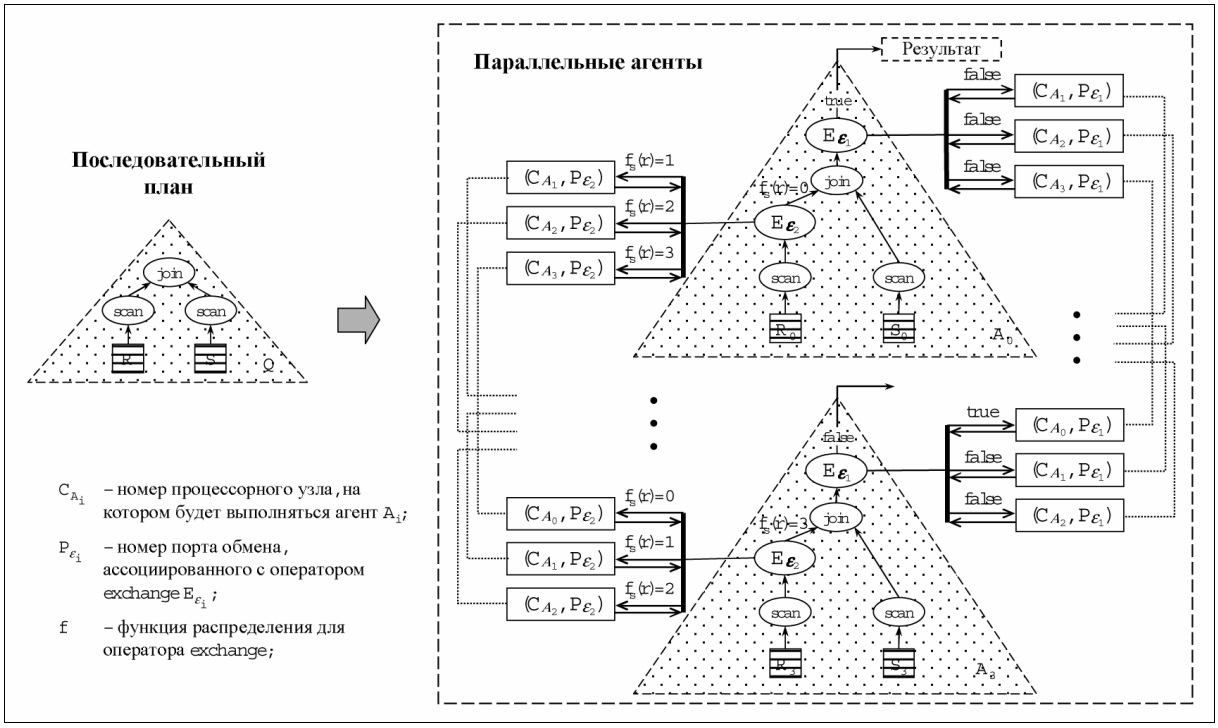

Рис. 14. Пример преобразования последовательного плана в параллельный

2. Формы параллелизма

Данный раздел посвящен классификации форм параллельной обработки транзакций. Рассматриваются межтранзакционный и внутритранзакционный параллелизм, межзапросный и внутризапросный параллелизм, межоперационный и внутриоперационный параллелизм, виды межоперационного параллелизма. Ссылки на литературу даются с использованием шифров из «Библиографического каталога по программированию и базам данных» [11].

Классификация различных форм параллелизма схематично изображена на Рис. 15. Прежде всего можно выделить *межтранзакционную* и *внутритранзакционную* формы параллелизма.

2.1 Межтранзакционный параллелизм

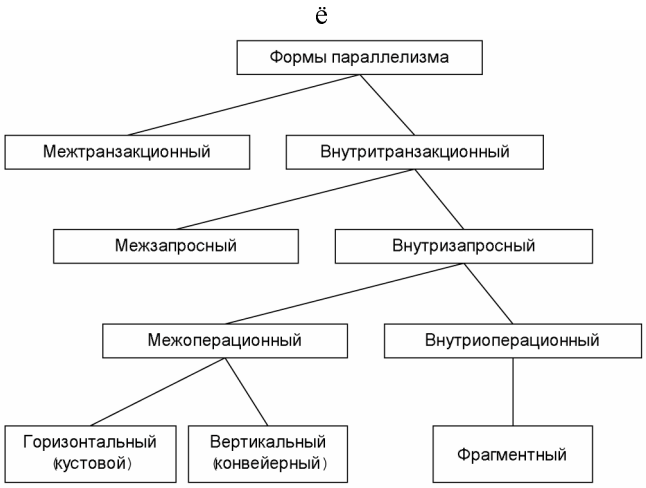

Рис. 15. Формы параллелизма

Межтранзакционный параллелизм подразумевает параллельное выполнение множества независимых транзакций над одной и той же базой данных. Данный вид параллелизма присутствует уже в однопроцессорных системах в виде так называемого многопользовательского режима и основан на перекрытии задержек ввода-вывода. Межтранзакционный параллелизм позволяет существенно увеличить суммарную производительность системы баз данных в режиме OLTP. Данный вид параллелизма также должен поддерживаться и в параллельной системе баз данных (наряду с внутритранзакционным параллелизмом), так как в противном случае мы получим очень плохое соотношение цена-производительность для режима OLTP [Pirahesh_90]. Для сокращения времени выполнения сложных транзакций необходимо использовать *внутритранзакционный параллелизм*.

2.2 Внутритранзакционный параллелизм

Внутритранзакционный параллелизм предполагает параллельное выполнение отдельной транзакции. Этот вид параллелизма может быть реализован либо в форме *межзапросного параллелизма*, либо в форме *внутризапросного параллелизма*.

2.3 Межзапросный (межоператорный) параллелизм

Межзапросный (межоператорный) параллелизм предполагает параллельное выполнение отдельных SQL операторов, принадлежащих одной и той же транзакции. Степень межзапросного параллелизма, однако, ограничена как количеством SQL операторов (запросов), составляющих данную транзакцию, так и ограничениями предшествования между отдельными SQL операторами.

Межзапросный параллелизм не поддерживается большинством современных СУБД, так как это потребовало бы от программиста явной спецификации межзапросных зависимостей с помощью некоторых специальных языковых конструкций.

2.4 Внутризапросный (внутриоператорный) параллелизм

Внутризапросный (внутриоператорный) параллелизм предполагает параллельное выполнение отдельного SQL оператора (запроса). Данная форма параллелизма характерна для реляционных систем баз данных. Это обусловлено тем, что реляционные операции над наборами кортежей по своей природе хорошо приспособлены для эффективного распараллеливания. Внутризапросный параллелизм реализуется оптимизатором запросов прозрачным для пользователя образом [Bultzingsloewen_89]. Для каждого запроса оптимизатор генерирует план выполнения запроса, который представляется в виде дерева (ациклического ориентированного графа), узлы которого соответствуют реляционным операциям, дуги - потокам данных между операциями, а в качестве листьев фигурируют отношения [Boral_82, Jarke_84]. Внутризапросный параллелизм может реализовываться либо в виде *межоперационного параллелизма*, либо в виде *внутриоперационного параллелизма*.

2.5 Межоперационный параллелизм

Межоперационный параллелизм предполагает параллельное выполнение реляционных операций, принадлежащих одному и тому же плану запроса. Межоперационный параллелизм может реализовываться либо в виде *горизонтального параллелизма*, либо в виде *вертикального параллелизма*.

2.6 Горизонтальный (кустовой) параллелизм

Горизонтальный (кустовой) параллелизм предполагает параллельное выполнение независимых поддеревьев дерева, представляющего план запроса. Основная проблема, связанная с кустовым параллелизмом, заключается в том, что очень трудно обеспечить, чтобы два подплана одного плана начали генерировать выходные данные в правильное время и в правильном темпе. При этом правильное время далеко не всегда означает одинаковое время, например для входных потоков операции хеш-соединения, а правильный темп далеко не всегда означает одинаковый темп, например для случая, когда входные потоки соединения слиянием имеют различные размеры [Graefe_93]. В силу указанных причин кустовой параллелизм редко используется на практике. В научных публикациях кустовой параллелизм исследовался главным образом в контексте оптимизации запросов с мультисоединениями [Chen_92, Chen_96, Lu_91, Schneider_90].

2.7 Вертикальный (конвейерный) параллелизм

Вертикальный (конвейерный) параллелизм предполагает организацию параллельного выполнения различных операций плана запроса на базе механизма конвейеризации. В соответствие с данным механизмом между смежными операциями в дереве запроса организуется поток данных в виде *конвейера*, по которому элементы данных (*гранулы*) передаются от *поставщика* к *потребителю*. Традиционный подход к организации конвейерного параллелизма заключается в использовании абстракции итератора для реализации операций в дереве запроса [Graefe_93]. Подобный подход впервые был использован при реализации System R [Selinger_79] и получил называние *синхронного конвейера* [Pirahesh_90].

Основным недостатком синхронного конвейера является блокирующий характер операций конвейерной обработки отдельных гранул. Если некоторая операция задерживается с обработкой очередной гранулы данных, то она блокирует работу всего конвейера. Для преодоления указанного недостатка может быть использован *асинхронный конвейер*, в котором поставщик и потребитель работают независимо друг от друга, а данные передаются через некоторый буфер. Поставщик помещает производимые гранулы в буфер, а потребитель забирает гранулы из данного буфера в соответствующем порядке. При этом необходимо определенное управление потоком данных, которое препятствовало бы пере-

полнению указанного буфера в случае, когда потребитель работает медленнее, чем поставщик. Подобный подход был использован в параллельной СУБД Volcano [Graefe_90] и в распределенной СУБД R* [Lohman_85]. Различные механизмы реализации синхронных и асинхронных конвейеров в контексте распределенных баз данных были исследованы в работе [SuMLC_86]. Следует отметить, что степень конвейерного параллелизма в любом случае ограничена количеством операций, вовлекаемых в конвейер. При этом для реляционных систем баз данных длина конвейера редко превышает 10 операций [DeWitt_92]. Поэтому для достижения более высокой степени распараллеливания наряду к конвейерным параллелизмом необходимо использовать *внутриоперационный параллелизм*.

2.8 Внутриоперационный параллелизм

Внутриоперационный параллелизм реализуется в основном в форме *фрагментного параллелизма* [DeWitt_92]. Некоторые авторы (см., например, [Rahm_93]) рассматривают и другие формы внутриоперационного параллелизма, базирующиеся на делении операции на подоперации. Однако данные формы параллелизма концептуально ничем не отличаются от рассмотренных выше и на практике большого значения не имеют.

2.9 Фрагментный параллелизм

Фрагментный параллелизм предполагает *фрагментацию* (разбиение на непересекающиеся части) отношения, являющегося аргументом реляционной операции [Graefe_93]. Одиночная реляционная операция выполняется в виде нескольких параллельных процессов (*агентов*), каждый из которых обрабатывает отдельный фрагмент отношения. Получаемые результирующие фрагменты сливаются в общее результирующее отношение [DeWitt_92].

В реляционных системах баз данных фрагментация подразделяется на *вертикальную* и *горизонтальную* [Williams_98]. *Вертикальная фрагментация* подразумевает разбиение отношения на фрагменты по столбцам (атрибутам). *Горизонтальная фрагментация* подразумевает разбиение отношения на фрагменты по строкам (кортежам). Практически все параллельные СУБД, поддерживающие фрагментный параллелизм, используют только горизонтальную фрагментацию. Поэтому везде ниже мы будем рассматривать только горизонтальную фрагментацию.

Теоретически фрагментный параллелизм способен обеспечить сколь угодно высокую степень распараллеливания реляционных операций. Однако на практике степень фрагментного параллелизма может быть существенно ограничена следующими двумя факторами. Во-первых, фрагментация отношения может зависеть от семантики операции. Например, операция соединения одних и тех же отношений по разным атрибутам требует различной фрагментации. Однако повторное разбиение фрагментированного отношения на новые фрагменты и распределение полученных фрагментов по процессорным узлам может быть связано с очень большими накладными расходами. Во-вторых, перекосы в распределении значений атрибутов фрагментации могут привести к значительным перекосам в размерах фрагментов и, как следствие, к существенному дисбалансу в загрузке процессоров.

3. Требования к параллельной системе баз данных

В данном разделе обсуждаются требования к параллельной системе баз данных. Рассматриваются следующие вопросы: масштабируемость, ускорение и расширяемость; балансировка загрузки, межпроцессорные коммуникации, когерентность кэшей, организация блокировок; коэффициент доступности базы данных, аппаратная отказоустойчивость, восстановление целостности базы данных после сбоя, оперативное восстановление базы данных, прозрачность для пользователя процессов восстановления системы.

Параллельная система баз данных должна представлять собой аппаратно-программный комплекс, способный хранить большой объем данных и обеспечивать эффективную обработку большого количества параллельных транзакций в режиме "24 часа в сутки, 7 дней в неделю". Другими словами, параллельная система баз данных должна представлять собой *систему высокой готовности*, то есть СУБД должна быть готова в любой момент обеспечить оперативную обработку запроса пользователя. В соответствие с выше сказанным можно сформулировать следующие *основные требования* к параллельной системе баз данных [Барон 95, Kim 84, Stonebraker 86, Valduriez 93]:

- высокая масштабируемость;
- высокая производительность;
- высокая доступность данных.

3.1 Масштабируемость

Важным свойством параллельных платформ является возможность их динамического наращивания в целях адаптации к увеличивающемуся размеру базы данных или возрастающим требованиям производительности [Valduriez 93]. Это достигается путем постепенного добавления в конфигурацию системы дополнительных процессоров, модулей памяти дисков и других аппаратных компонент. Данный процесс называется *масштабированием* системы. При удвоении аппаратной мощности системы мы можем ожидать, что ее производительность также возрастет вдвое, однако на практике реальное приращение производительности часто оказывается существенно ниже. Например, масштабируемость *SMP* систем ограничена 20-30 процессорами [Valduriez 93]. При дальнейшем наращивании *SMP* системы производительность возрастает очень слабо или даже начинает происходить ее падение [HuaLP 91]. Это связано с тем, что процессоры начинают все более простаивать в ожидании доступа к разделяемым ресурсам (общая шина доступа к памяти и дискам) . В соответствие с этим масштабируемость любой многопроцессорной системы определяется *эффективностью распараллеливания*.

Существуют две основные качественные характеристики эффективности распараллеливания: *ускорение* и *расширяемость*. Дадим следующее формализованное определение указанных понятий, следуя работе [DeWitt 92].

3.2 Ускорение

Пусть даны две различные конфигурации *A* и *B* параллельной машины баз данных с заданной архитектурой, различающиеся количеством процессоров и ассоциированных с ними устройств (мы предполагаем, что все конфигурации предполагают пропорциональное наращивание модулей памяти и дисков). Пусть задан некоторый тест *Q*. *Коэффициент ускорения* a_{AB}, получаемый при переходе от конфигурации *A* к конфигурации *B*, определяется следующей формулой

$$a_{AB} = \frac{d_A t_{QA}}{d_B t_{QB}}.$$

Здесь d_A - *степень параллелизма* (количество процессоров) конфигурации A; d_B - степень параллелизма конфигурации B; t_{QA} - время, затраченное конфигурацией A на выполнение теста Q; t_{QB} - время, затраченное конфигурацией B на выполнение теста Q.

Говорят, что система демонстрирует *линейное ускорение*, если коэффициент ускорения остается равным единице для всех конфигураций данной системы.

3.3 Расширяемость

Пусть теперь задан набор тестов Q_1, Q_2, ..., количественно превосходящих некоторый фиксированный тест Q в i раз, где i - номер соответствующего теста. Пусть заданы конфигурации параллельных машин баз данных A_1, A_2, ..., превосходящие по степени параллелизма некоторую минимальную конфигурацию A в j раз, где j - номер соответствующей конфигурации. Тогда *коэффициент расширяемости* e_{km}, получаемый при переходе от конфигурации A_k к конфигурации A_m ($k < m$), задается формулой

$$e_{km} = \frac{t_{Q_k A_k}}{t_{Q_m A_m}}.$$

Говорят, что система демонстрирует *линейную расширяемость*, если коэффициент расширяемости остается равным единице для всех конфигураций данной системы.

Говорят, что параллельная система *хорошо масштабируема*, если она демонстрирует ускорение и расширяемость, близкие к линейным.

Ускорение позволяет определить эффективность наращивания системы на сопоставимых задачах. Расширяемость позволяет измерить эффективность наращивания системы на бóльших задачах.

Основным фактором, препятствующим хорошей масштабируемости системы, являются помехи, возникающие при конкурентном доступе процессоров к разделяемым ресурсам.

3.4 Производительность

Производительность параллельной системы баз данных определяющим образом зависит от эффективного решения следующих ключевых проблем:

- межпроцессорные коммуникации;
- когерентность КЭШей;
- организация блокировок;
- балансировка загрузки.

Межпроцессорные коммуникации в параллельных системах баз данных обычно порождают трафик значительного объема, что может выражаться в высоких накладных расходах, связанных с передачей сообщений от одного процессора - другому.

Когерентность КЭШей. Учитывая, что обращение к диску примерно в 10^5-10^6 раз медленнее, чем обращение к оперативной памяти, мы можем существенным образом повысить общую производительность системы баз данных, используя кэширование страниц диска в оперативной памяти. При этом, если один и тот же фрагмент базы данных кэширован в приватной памяти различных процессоров, то мы должны *согласовывать* изменения данного фрагмента в кэшах различных процессоров, то есть обеспечивать *когерентность кэшей*. Поддержание когерентности кэшей в многопроцессорной системе может быть связано с серьезными накладными расходами.

Организация блокировок. Если различные процессоры обрабатывают одни и те же объекты базы данных (отношения, кортежи и др.), нам необходимо поддерживать *глобальную таблицу блокировок*, используемую всеми процессорами. Это может выражаться

в больших накладных расходах.

Балансировка загрузки процессоров является одной из ключевых проблем для обеспечения высокой эффективности параллельной обработки запросов. СУБД должна разбивать запрос на параллельные агенты и распределять их по процессорам таким образом, чтобы обеспечивалась равномерная загрузка всех задействованных процессоров. Особенно остро вопрос с балансировкой загрузки возникает при использовании фрагментного параллелизма. Фактором, который может существенным образом снизить эффективность распараллеливания реляционных операций, особенно соединения и сортировки, является величина *перекоса*, присутствующая в данных, подлежащих обработке. Исследования показали, что в реальных базах данных некоторые значения для определенного атрибута встречаются значительно чаще, чем остальные [Christodoulakis 83, Lynch 88, MontgomeryDL 83]. В частности, Линч (Lynch) отмечает в [Lynch 88], что для текстовых атрибутов характерно распределение значений по закону Зипфа [Zipf 49]. Подобная неоднородность называется *перекосом значений атрибута* [WaltonDJ 91]. Лакшми (Lakshmi) и Ю (Yu) показали [LakshmiY 90], что при наличии перекоса данных ускорение, достигаемое при параллельным выполнении операций соединения, может быть ограничено катастрофическим образом по причине перегрузки одних процессоров и недогрузки других.

3.5 Доступность данных

Одной из критических характеристик параллельных систем баз данных является способность системы обеспечить высокую степень доступности данных в условиях отказа некоторых аппаратных компонент. Вероятность отказа аппаратуры в однопроцессорной системе не велика. Однако в с системе с тысячами процессорных узлов данная вероятность возрастает в тысячи раз. Поэтому проблема обеспечения высокой доступности данных в многопроцессорных системах имеет важное значение.

Коэффициент доступности базы данных нестрого может быть определен как отношение межу промежутком времени, в течение которого база данных была действительно доступна пользователям, и промежутком времени, в течение которого пользователи требовали доступ к базе данных. Например, если пользователи требовали доступ к базе данных в течение 8 часов в день, а реально база данных была доступна только в течение 6 часов, то коэффициент доступности составляет 6/8 = 0.75 в течение 8-часового периода. Систему баз данных с *высокой доступностью данных* можно определить как систему, обеспечивающую прием запросов пользователей 24 часа в сутки с коэффициентом доступности не менее 0.99 [Kim 84].

Высокая доступность данных определяется следующими четырьмя факторами [Kim 84]:

1. аппаратная отказоустойчивость;
2. восстановление целостности базы данных после сбоя;
3. оперативное восстановление базы данных;
4. прозрачность для пользователя процессов восстановления системы.

Аппаратная отказоустойчивость является основным фактором обеспечения высокой доступности данных в параллельных системах баз данных с большим количеством процессорных узлов. Под аппаратной отказоустойчивостью понимают сохранение общей работоспособности системы при одиночном отказе таких аппаратных компонент, как процессор, модуль памяти, диск, и каналов доступа к перечисленным компонентам [Kim 84]. В частности, одиночный отказ любого устройства не должен привести к потере целостности базы данных и тем более к физической утрате какой-то части базы данных.

Восстановление целостности базы данных после сбоя предполагает поддержку ACID[1)] транзакций и журнализацию изменений [Кузнецов 01a, Gray 78]. Данные возможности поддерживаются большинством современных СУБД с архитектурой клиент-сервер.

Оперативное восстановление базы данных предполагает восстановление нормальной работоспособности системы с сохранением режима обслуживания пользователей. При этом коэффициент доступности данных может временно уменьшаться.

Прозрачность для пользователя процессов восстановления системы предполагает незначительное уменьшение коэффициента доступности базы данных во время сбоя и последующего восстановления. Сложность данной проблемы заключается в том, что выход из строя одного из дисков может привести к серьезному дисбалансу загрузки процессоров, например, в силу удвоения нагрузки на узел, содержащий копию утраченных данных. Возможное решение указанной проблемы заключается в том, чтобы фрагментировать копию диска по другим дискам таким образом, чтобы она допускала параллельный доступ [HsiaoD 93].

1) Транзакция, характеризующаяся следующими свойствами: атомарность, согласованность, изолированность, долговечность - ACID (Atomicity, Consistency, Isolation, Durability).

4. Классификация и сравнительный анализ архитектур параллельных систем баз данных

Данный раздел посвящен классификации и сравнительному анализу архитектур параллельных систем баз данных. Рассматриваются архитектуры с разделяемой памятью и дисками, архитектуры с разделением дисков, архитектуры без совместного использования, иерархические и гибридные архитектуры.

4.1 Классификация Стоунбрейкера

Наиболее распространенной системой классификации параллельных систем баз данных является система, предложенная Майклом Стоунбрейкером (Michael Stonebraker) в работе [Stonebraker 86]. Схематично данная классификация изображена на Рис. 16. Здесь **P** обозначает процессор, **M** – модуль оперативной памяти, **D** – дисковое устройство, **N** - соединительную сеть.

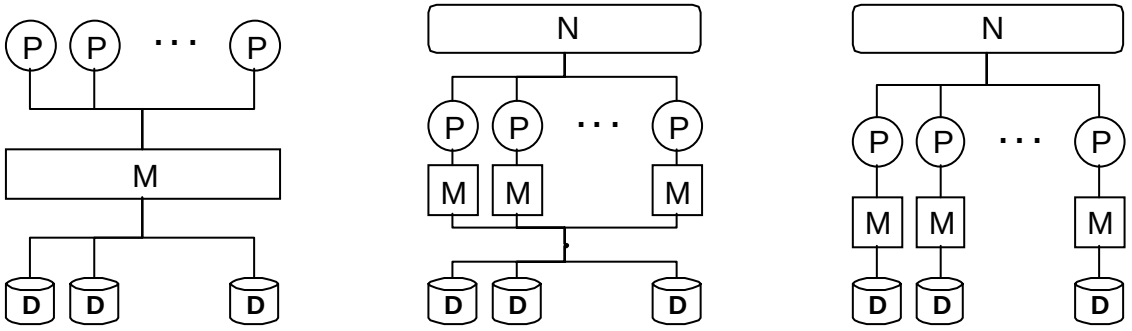

Рис. 16. Классификация Стоунбрейкера

В соответствие с классификацией Стоунбрейкера параллельные системы баз данных могут быть разделены на следующие три базовых класса в зависимости от способа разделения аппаратных ресурсов:

(а) *SE* (*Shared-Everything*) - архитектура с разделяемыми памятью и дисками;

(б) *SD* (*Shared-Disks*) - архитектура с разделяемыми дисками;

(в) *SN* (*Shared-Nothing*) - архитектура без совместного использования ресурсов.

SE архитектура (в работе [Stonebraker 86] эта архитектура обозначается как *Shared-Memory*) представляет системы баз данных, в которых все диски напрямую доступны всем процессорам с одинаковым временем доступа и все процессоры разделяют общую оперативную память. Межпроцессорные коммуникации в *SE* системах осуществляются через общую оперативную память. Доступ к дискам в *SE* системах обычно осуществляется через общий буферный пул. При этом следует отметить, что каждый процессор в *SE* системе имеет собственную кэш-память.

Существует большое количество параллельных систем баз данных с *SE* архитектурой. По существу все ведущие коммерческие СУБД сегодня имеют реализацию на базе *SE* архитектуры. В качестве одного из первых примеров портирования с однопроцессорной системы на *SE* архитектуру можно привести реализацию DB2 на IBM3090 с 6 процессорами [Cheng 84]. Другим примером является параллельное построение индексов в Informix OnLine 6.0 [Davison 92]. Следует отметить, однако, что подавляющее большинство коммерческих *SE* систем использует только межтранзакционный параллелизм (то есть

внутритранзакционный параллелизм отсутствует). Тем не менее, к настоящему моменту создано несколько исследовательских прототипов *SE* систем, использующих внутризапросный параллелизм, например, XPRS [Stonebraker 88], DBS3 [Bergsten 91, Bergsten 93a] и Volcano [Graefe 90, Graefe 94].

Базовой аппаратной платформой для реализации систем с *SE* архитектурой обычно служит SMP, хотя потенциально *SE* системы можно строить на платформах с архитектурой NUMA и даже MPP с виртуально общей, физически распределенной памятью [Amza 96].

SD архитектура представляет системы баз данных, в которых любой процессор имеет доступ к любому диску, однако каждый процессор имеет свою приватную оперативную память [Rahm 93]. Процессоры в таких системах соединены посредством некоторой высокоскоростной сети, позволяющей осуществлять передачу данных. Примерами параллельных систем баз данных с *SD* архитектурой являются IBM IMS [StricklandUW 82], Oracle Parallel Server [Linder 93] на nCUBE [Дубова 95] и VAXcluster [KronenbergLS 86], IBM Parallel Sysplex [King 97, Nick 97] и др.

SN архитектура характеризуется наличием у каждого процессора собственной оперативной памяти и собственного диска. Как и в *SD* системах, процессорные узлы соединены некоторой высокоскоростной сетью, позволяющей организовывать обмен сообщениями между процессорами [DeWitt 92]. К настоящему моменту создано большое количество исследовательских прототипов и несколько коммерческих систем с *SN* архитектурой, использующих фрагментный параллелизм. В качестве примеров исследовательских прототипов *SN* систем можно привести следующие системы: ARBRE [Lorie 89], BUBBA [Boral 90], EDS [Skelton 92], GAMMA [DeWitt 90], KARDAMOM [Bultzingsloewen 88], PRISMA [Apers 92]. Примерами коммерческих систем с *SN* архитектурой являются NonStop SQL [Хаманн 97, Chambers 93, EnglertGH 95], Informix PDQ [Clay 93], NCR (Teradata) DBC/1012 [ЛисянскийС 97, Page 92], IBM DB2 PE [Baru 95] и др.

4.2 Расширение классификации стоунбрейкера

Копеланд (Copeland) и Келлер (Keller) предложили в работе [CopelandK 89] *расширение классификации Стоунбрейкера* путем введения двух дополнительных классов архитектур параллельных машин баз данных (Рис. 17):

(г) *CE* (*Clustered-Everything*) - архитектура с *SE* кластерами, объединенными по принципу *SN*;

(д) *CD* (*Clustered-Disk*) - архитектура с *SD* кластерами, объединенными по принципу *SN*. Граница *SD* кластеров на Рис. 17 распространены на общую (*глобальную*) соединительную сеть, так как в них может присутствовать собственная (*локальная*) соединительная сеть.

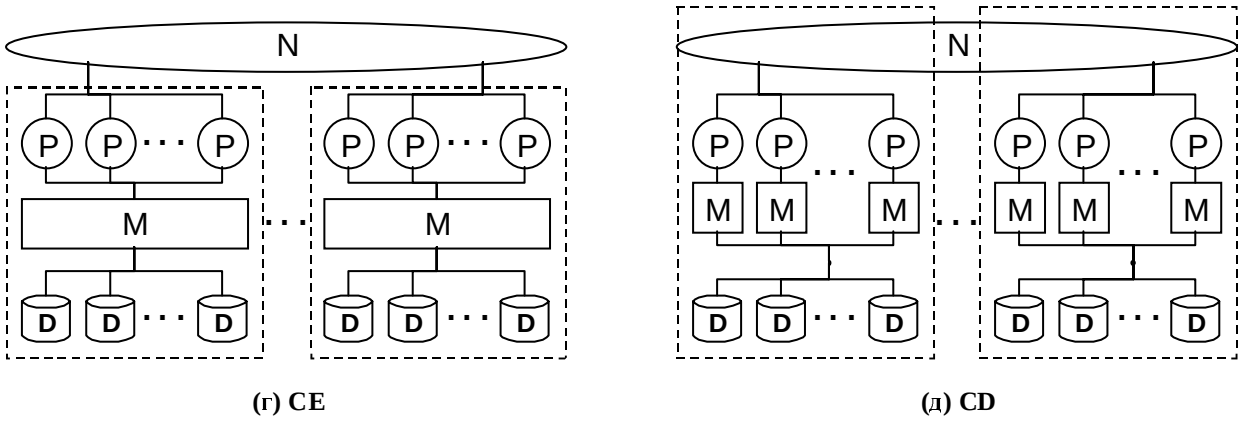

Рис. 17. Расширение классификации Стоунбрейкера

Эти архитектуры также получили название *иерархических* [Graefe_93]. На Рис. 17 изображены двухуровневые иерархии. Однако классификационный подход Копеланда и Келлера легко может быть распространен на архитектуры с тремя и более уровнями иерархии. В качестве примера можно привести трехуровневую иерархическую архитектуру CD_2 (Clustered-Disk with 2-processor modules. Данная архитектура была использована при проектировании системы Омега [Sokolinsky_01]. Двухуровневые иерархические архитектуры были исследованы в целом ряде работ (см., например, [BouganimFV_96, CopelandK_89, HuaLP_91, PramanikT_97, XuD_97]). Трехуровневые иерархические архитектуры до последнего времени оставались мало изученными.

4.3 Гибридная архитектура C_D^N

Эрхард Рам (Erhard Rahm) в работе [Rahm_92] предложил рассматривать *гибридные архитектуры*. Гибридные архитектуры нельзя отнести ни к одному из выше описанных классов. В качестве примера гибридной архитектур можно привести архитектуру C_D^N. Данная архитектура была описана в работах [Соколинский_01, Sokolinsky_01]. C_D^N архитектура строится как набор однотипных *SD* кластеров, объединенных по принципу *SN*. Отличительной особенностью данной системной архитектуры является то, что на верхних уровнях системной иерархии *SD* кластеры рассматриваются как *SN* системы (см. Рис. 18). Это выражается в том, что каждому процессорному узлу *логически* назначается отдельный диск. Такой подход позволяет избежать проблем, связанных с реализацией глобальной таблицы блокировок и поддержкой когерентности кэшей, характерных для *SD* систем [Valduriez_93], и одновременно использовать преимущества *SD* архитектуры в плане возможности балансировки загрузки. Подобный подход был также использован при разработке параллельной системы баз данных NonStop SQL/MP [EnglertGH_95].

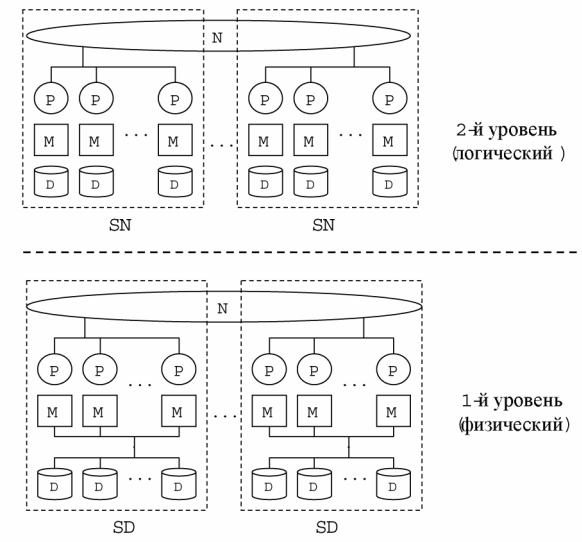

Рис. 18. Гибридная архитектура C_D^N

4.4 Сравнительный анализ архитектур

Сравнительный анализ *SE*, *SD* и *SN* архитектур был выполнен Стоунбрейкером в классической работе [Stonebraker_86]. Этот анализ показал, что для масштабируемых высокопроизводительных систем баз данных из трех указанных архитектур наиболее предпочтительной является *SN* архитектура.

Табл. 1. Сравнение архитектур

	SE	***SD***	***SN***	***CE***	***CD***	C_D^N
Масштабируемость	0	1	2	3	3	3
Доступность данных	0	1	3	1	2	2
Баланс загрузки	3	2	0	2	1	1
Межпроцессорные коммуникации	3	0	0	2	1	1
Когерентность кэшей	2	0	3	2	0	3
Организация блокировок	2	0	3	2	0	3
Сумма баллов	**10**	**4**	**11**	**12**	**7**	**13**

В Табл. 1 приведен сравнительный анализ шести различных архитектур параллельных систем баз данных по критериям, непосредственно вытекающим из требований к параллельной системе баз данных. Для оценки показателей используется трехбалльная система: 0 - «плохо»; 1 - «удовлетворительно», 2 - «хорошо», 3 – «превосходно».

Масштабируемость *SE* архитектуры характеризуется как низкая. Это связано с тем, что при количестве процессорных узлов больше 30 общая шина доступа к памяти и дискам становится узким местом. *SD* архитектура демонстрирует среднюю масштабируемость, так как общая шина доступа к дискам по-прежнему остается узким местом. *SN*, *CD*, *CE* и C_D^N архитектуры демонстрируют хорошую масштабируемость.

Доступность данных для *SN* архитектуры можно классифицировать как среднюю. Это связано с тем, что страховочные копии в *SN* системе должны фрагментироваться по многим узлам [HsiaoD_93] для того, чтобы в случае отказа одного из дисков его страховочная копия была доступна в параллельном режиме (в противном случае может возникнуть серьезный дисбаланс загрузки). Поддержание когерентности фрагментированных страховочных копий потребует определенных накладных расходов, связанных прежде всего с пересылкой большого количества данных по соединительной сети. *SE* и *CE* архитектуры характеризуется "плохой" доступностью данных из-за низкой аппаратной отказоустойчивости *SE* кластера [Pfister_98]. SD, CD и C_D^N архитектуры демонстрируют наилучшую доступность данных, так как все проблемы, связанные с обеспечением высокой доступности, могут быть эффективно решены на уровне отдельных *SD* кластеров.

Баланс загрузки для *SN* архитектуры является серьезной проблемой, так как *SN* системы весьма чувствительны к перекосу данных [LakshmiY_90]. Иерархические архитектуры типа *CE*, *CD* и C_D^N позволяют достичь лучшей сбалансированности загрузки, так как баланс загрузки может здесь выполняться на двух уровнях - межкластерном и внутрикластерном. *SD* кластеры позволяют достичь удовлетворительной балансировки загрузки, поскольку каждому процессору доступны все диски. Наилучший баланс загрузки обеспечивают *SE* кластеры, так как помимо дисков процессорам доступна вся оперативная память [Valduriez_93].

Высокая стоимость ***межпроцессорных коммуникаций*** является одной из слабых сторон *SN* архитектуры [EnglertGH_95, Stonebraker_86]. Использование *CE* архитектуры позволяет значительно сократить накладные расходы, связанные с межпроцессорными коммуникациями [HuaLP_91]. Межпроцессорные коммуникации на уровне *SE* кластера могут быть реализованы очень эффективно через разделяемую память. *CD* и C_D^N архитектуры уступают по этому показателю *CE* архитектуре, однако могут превзойти *SN* архитектуру, так как потенциально внутрикластерные коммуникации могут быть реализованы более эффективно, чем межкластерные [Sokolinsky_99, Sokolinsky_99a].

Когерентность кэшей является серьезной проблемой для *CD* архитектуры, так как в *SD* кластере одни и те же страницы разделяемых дисков буферизуются в индивидуальных модулях памяти. Копии страниц остаются в буферных пулах некоторое время после завершения обратившейся к ним транзакции, поэтому изменения, сделанные одним узлом *SD* кластера, могут быть аннулированы изменениями, сделанными другим узлом *SD* кластера [Rahm_93]. *CE* архитектура имеет лучшие показатели по данному параметру

по сравнению с *CD* архитектурой, так как *SE* кластеры используют общий буферный пул в разделяемой памяти. Однако *CE* архитектура уступает по этому параметру *SN* архитектуре, так как в *SE* кластере необходимо поддерживать когерентность данных в индивидуальных кэш-памятях процессоров [Pfister_98]. Отметим, что C_D^N архитектура свободна от указанного недостатка, так как на виртуальном уровне в ней отсутствует разделение ресурсов. В *SN* системах проблема когерентности так же отсутствует, так как у них нет разделяемых ресурсов.

Еще одной серьезной проблемой для *CD* архитектуры являются трудности с ***организацией блокировок*** объектов базы данных со стороны обращающихся к ним транзакций. В *SD* кластере на каждом процессорном узле необходимо поддерживать копию глобальной таблицы блокировок, что может потребовать серьезных накладных расходов [MohanN_92]. *CE* архитектура в значительной мере избавлена от этой проблемы, так как глобальная таблица блокировок *SE* кластера хранится в единственном экземпляре в разделяемой оперативной памяти. В *SN* системах нет необходимости в поддержании глобальной таблицы блокировок по причине отсутствия разделения ресурсов, поэтому *SN* архитектура занимает наилучшую позицию по данному параметру. C_D^N архитектура в полной мере наследует это качество от *SN* архитектуры.

Исходя из проведенного анализа, мы можем сделать ***вывод***, что нет весомых причин для поддержки *CD* архитектуры в чистом виде. *CE* архитектура выглядит более привлекательно, чем *SE* архитектура. Однако, если принимать во внимание требования к параллельной системе баз данных высокой готовности, наилучшим выбором является C_D^N архитектура.

Литература

1. *Девитт Д., Грэй Д.* Параллельные системы баз данных: будущее высоко эффективных систем баз данных // СУБД. -1995. -№2. -С. 8-31. [http://www.osp.ru/dbms/1995/02/21.htm]
2. *Корнеев В.В., Гареев А.Ф., Васютин С.В., Райх В.В.* Базы данных. Интеллектуальная обработка информации. 2-е издание. -М.: Нолидж, 2001. -496 с.
3. *Оззу М.Т., Валдуриз П.* Распределенные и параллельные системы баз данных // СУБД. -1996. -№4. -С. 4-26. [http://www.osp.ru/dbms/1996/04/4.htm]
4. *Pfister G.* Sizing Up Parallel Architectures // DataBase Programming & Design OnLine [http://www.dbpd.com]. -May 1998. -Vol. 11, No. 5.
5. *Соколинский Л.Б.* Организация параллельного выполнения запросов в многопроцессорной машине баз данных с иерархической архитектурой // Программирование. -2001. -№6. -С. 13-29.
6. *Соколинский Л.Б.* Параллельные машины баз данных // Природа. Естественно-научный журнал Российской академии наук. -2001. -№8. -С. 10-17. [http://www.ibmh.msk.su/vivovoco/VV/JOURNAL/NATURE/08_01/PARBASE.HTM]
7. *Соколинский Л.Б.* Методы организации параллельных систем баз данных на вычислительных системах с массовым параллелизмом: Дис. ... докт. физ.-мат. наук: 05.13.18 / Челябинский государственный университет. -Челябинск, 2003. -247 л. [http://www.csu.ru/~sok/dissertation/dissert.pdf]
8. *Ульман Дж., Уидом Дж.* Введение в системы баз данных. -М.: ЛОРИ, 2000. -347 с.
9. *Garcia-Molina H., Ullman J.D., Widom J.* Database System Implementation. -Prentice Hall, 2000. -653 p.
10. *Graefe G.* Query evaluation techniques for large databases // ACM Computing Surveys. -1993. -Vol. 25, No. 2. -P. 73-169.

Информационные ресурсы сети Интернет

11. Библиографический каталог по программированию и базам данных [http://reindeer.csu.ac.ru/oracle/bibl]
12. Словарь-справочник по программированию и базам данных [http://reindeer.csu.ac.ru/oracle/dict]
13. Сервер информационных технологий CITForum.ru [http://www.citforum.ru]
14. ACM SIGMOD [http://www.acm.org/sigmod]
15. DBLP (Computer science bibliography) [http://www.informatik.uni-trier.de/~ley/db]