ПАРАЛЛЕЛЬНЫЕ СУБД

Е.В. АБРАМОВ, Д.О. ШАГЕЕВ

2008 г.

РЕЛЯЦИОННЫЕ БАЗЫ ДАННЫХ. ОСОБЕННОСТИ ПАРАЛЛЕЛЬНЫХ СУБД

Процессы обработки структурированных и неструктурированных данных (медиа данные) в больших объемах немыслимы без использования современных параллельных систем управления базами данных (СУБД). На сегодняшний день существует около десятка коммерческих систем данного класса: Oracle RAC, NCR Teradata, IBM Informix OnLine, IBM DB2, Microsoft SQL Server и др. Как правило, подобные системы функционируют на параллельных архитектурах класса мейнфрейм (mainframe). Объемы внешней памяти таких систем измеряются десятками, сотнями и даже тысячами гигабайт. Стоимость подобных систем также впечатляет.

Вместе с тем появляются и некоммерческие программные продукты параллельной обработки данных. Они разрабатываются как открытые системы, предоставляющие исходные коды программного обеспечения под лицензией GNU. К числу таких систем можно отнести СУБД MySQL Cluster и PostgreSQL Cluster. Они представляют собой полноценные СУБД кластерного типа, реализующие все стандартные функции. Требования к аппаратному обеспечению таких систем значительно скромнее. Скромнее и возможности, но их вполне хватает для построения высокопроизводительных систем обработки информации среднего предприятия. Подобные системы могут быть созданы например на базе имеющихся компьютерных классов с привлечением небольшой группы специалистов.

Реляционная модель данных [1, 2]

Реляционная модель предложена сотрудником компании IBM Е.Ф.Коддом в 1970 г. В настоящее время эта модель является фактическим стандартом, на который ориентируются практически все современные коммерческие СУБД. Дадим определения некоторым базовым понятиям.

Декартово произведение: Для заданных конечных множеств $D_1,D_2, \ldots,D_n$ (не обязательно различных) декартовым произведением $D_1*D_2*\ldots*D_n$ называется множество произведений вида $d_1*d_2*\ldots*d_n$, где $d_1 \in D_1, d_2 \in D_2, .., d_n \in D_n$.

Пример. Пусть заданы два множества – A (a1,a2,a3) и B (b1,b2). Тогда их декартово произведение

C=A*B (a1*b1, a2*b1, a3*b1, a1*b2, a2*b2, a3*b2).

Отношение. Отношением R, определенным на множествах $D_1,D_2,\ldots,D_n$ называется подмножество декартова произведения $D_1*D_2*\ldots*D_n$. При этом:

- Множества D_1, D_2, …, D_n называются *доменами отношения*.
- Элементы декартова произведения $d_1*d_2*…*d_n$ называются *кортежами*.
- Число *n* определяет степень, или «арность» отношения (*n*=1 – унарное, *n*=2 – бинарное, ..., в общем случае – *n*-арное).
- Количество кортежей называется *мощностью отношения.*

Пример. На множестве C из предыдущего примера могут быть определены отношения R1 (a1*b1, a3*b2) или R2 (a1*b1, a2*b1, a1*b2).

Отношения удобно представлять в виде таблиц. На рис. 1 представлена таблица (отношение степени 5), содержащая некоторые сведения о работниках гипотетического предприятия.

	целое	*строка*		*целое*		←Типы данных
	номер	*имя*	*должность*	*деньги*		←Домены
Отношение →	**табельный номер**	**имя**	**должность**	**оклад**	**премия**	←Атрибуты
	2930	Иванов	инженер	112	40	
	2931	Петров	вед. инженер	144	50	←Кортежи
	2932	Сидоров	бухгалтер	92	35	
	↑ Ключ					

Рис. 1

Строки таблицы соответствуют кортежам. Каждая строка представляет собой описание одного объекта реального мира (в данном случае – работника), характеристики которого содержатся в столбцах. Можно провести аналогию между элементами реляционной модели данных и элементами модели «сущность-связь». Реляционные отношения соответствуют наборам сущностей, а кортежи – сущностям. Поэтому, как и в модели «сущность-связь», столбцы в таблице, представляющей реляционное отношение, называют *атрибутами*.

Каждый атрибут определен на домене. Поэтому домен можно рассматривать как множество допустимых значений данного атрибута.

Несколько атрибутов одного отношения и даже атрибуты разных отношений могут быть определены на одном и том же домене. В примере, показанном на рис. 1, атрибуты "*Оклад*" и "*Премия*" определены на домене "*Деньги*". Поэтому понятие домена имеет семантическую нагрузку: данные можно считать сравнимыми только тогда, когда они относятся к одному домену. Так, в рассматриваемом нами примере сравнение атрибутов "Табельный номер" и "Оклад" является семантически некорректным, хотя они и содержат данные одного типа.

Именованное множество пар "имя атрибута – имя домена" называется *схемой отношения*. Набор именованных схем отношений представляет собой *схему базы данных*.

Атрибут, значения которого однозначно идентифицирует кортежи, называется *ключевым* (или просто *ключом*). В нашем случае ключом является атрибут "*Табельный номер*", поскольку его значение уникально для каждого работника предприятия. Если кортежи идентифицируются только сцеплением

значений нескольких атрибутов, то говорят, что отношение имеет составной ключ.

Отношение может содержать несколько ключей. Но всегда один из ключей объявляется *первичным*, его значения не могут обновляться. Все остальные ключи отношения называются *возможными ключами*.

В отличие от иерархической и сетевой моделей данных, в реляционной модели отсутствует понятие группового отношения. Для отражения ассоциаций между кортежами разных отношений используется дублирование их ключей (рис. 2). Например связь между отношениями ОТДЕЛ и СОТРУДНИК создается путем копирования первичного ключа "Номер_отдела" из первого отношения во второе.

Таким образом:

• Для того, чтобы получить список работников данного подразделения, необходимо:

1. Из таблицы ОТДЕЛ установить значение атрибута "*Номер_отдела*", соответствующее данному "*Наименованию_отдела*".

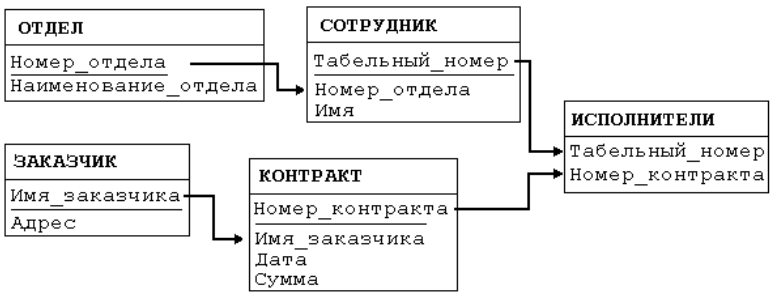

Рис. 2

2. Выбрать из таблицы СОТРУДНИК все записи, значение атрибута *"Номер_отдела"* которых равно полученному на предыдущем шаге.

• Для того, чтобы узнать в каком отделе работает сотрудник, нужно выполнить обратную операцию:

1. Определяем *"Номер_отдела"* из таблицы СОТРУДНИК.
2. По полученному значению находим запись в таблице ОТДЕЛ.

Атрибуты, представляющие собой копии ключей других отношений, называются *внешними ключами*.

Свойства отношений:

1. Отсутствие кортежей-дубликатов. Из этого свойства вытекает наличие у каждого кортежа первичного ключа. Для каждого отношения полный набор его атрибутов является первичным ключом. Однако при определении первичного ключа должно соблюдаться требование "минимальности", т.е. в такой ключ не должны входить те атрибуты, которые можно отбросить без ущерба для основного свойства первичного ключа – однозначно определять кортеж.
2. Отсутствие упорядоченности кортежей.

3. Отсутствие упорядоченности атрибутов. Для ссылки на значение атрибута всегда используется имя атрибута.

4. Атомарность значений атрибутов: среди значений домена не могут содержаться множества значений, т.е. отношения.

Операции над данными (реляционная алгебра)

Различают обработку кортежей и обработку отношений.

Операции обработки кортежей связаны с изменением состава кортежей в каком-либо отношении. К ним относятся:

ДОБАВИТЬ – надо задать имя отношения и ключ кортежа.

УДАЛИТЬ – необходимо указать имя отношения и идентифицировать кортеж или группу кортежей, подлежащих удалению.

ИЗМЕНИТЬ – выполняется для названного отношения и может корректировать как один, так и несколько кортежей.

Операции обработки отношений. На входе каждой такой операции используется одно или несколько отношений. Результатом выполнения операции всегда является новое отношение.

В рассмотриваемых ниже примерах (они заимствованы из [1]) используются следующие отношения:

P(D1,D2,D3)	Q(D4,D5)	R(M,P,Q,T)	S(A,B)
1 11 x	x 1	x 101 5 a	5 a
2 11 y	x 2	y 105 3 a	10 b
3 11 z	y 1	z 500 9 a	15 c
4 12 x		w 50 1 b	2 d
		w 10 2 b	6 a
		w 300 4 b	1 b

В реляционной алгебре определены следующие операций обработки отношений:

Проекция /PROJECT/. Суть этой операции состоит в том, что берется отношение R, удаляются некоторые из его компонентов и (или) переупорядочиваются оставшиеся компоненты.

Обозначение: $\pi_A(R)$. Пример:

$$\pi_{M,T}\begin{bmatrix} x & a \\ y & a \\ z & a \\ w & b \\ w & b \\ w & b \end{bmatrix} = \begin{bmatrix} x & a \\ y & a \\ z & a \\ w & b \end{bmatrix}$$

Выборка /SELECT/. На входе используется одно отношение, результат - новое отношение, построенное по той же схеме, содержащее подмножество кортежей исходного отношения, удовлетворяющих условию выборки.

Обозначение: $\sigma_{\theta(R)}$, θ - условие селекции. Пример:

$$\sigma_{\text{Â=11}}(\mathrm{R}) = \begin{bmatrix} 1 & 11 & x \\ 2 & 11 & y \\ 3 & 11 & z \end{bmatrix}$$

Объединение /UNION/. Отношения-операнды в этом случае должны быть определены по одной схеме. Результирующее отношение содержит все строки операндов, за исключением повторяющихся. Обозначение: $\mathrm{R}_1 \cup \mathrm{R}_2$. Пример:

$$\sigma_{\mathrm{Q,T}}(\mathrm{R}) \cup \mathrm{S} = \begin{bmatrix} 5 & a \\ 3 & a \\ 9 & a \\ 1 & b \\ 2 & b \\ 4 & b \end{bmatrix} \cup \begin{bmatrix} 5 & a \\ 10 & b \\ 15 & c \\ 2 & d \\ 6 & a \\ 1 & b \end{bmatrix} = \begin{bmatrix} 5 & a \\ 3 & a \\ 9 & a \\ 1 & b \\ 2 & b \\ 4 & b \\ 10 & b \\ 15 & c \\ 2 & d \\ 6 & a \end{bmatrix}$$

*Пересечение /*INTERSECT*/*. На входе операции – два отношения, определенные по одной схеме. На выходе – отношение, содержащие кортежи, которые присутствуют в обоих исходных отношениях. Обозначение: $\mathrm{R}_1 \cap \mathrm{R}_2$. Пример:

$$\sigma_{\mathrm{Q,T}}(\mathrm{R}) \cap \mathrm{S} = \begin{bmatrix} 5 & a \\ 3 & a \\ 9 & a \\ 1 & b \\ 2 & b \\ 4 & b \end{bmatrix} \cap \begin{bmatrix} 5 & a \\ 10 & b \\ 15 & c \\ 2 & d \\ 6 & a \\ 1 & b \end{bmatrix} = \begin{bmatrix} 5 & a \\ 1 & b \end{bmatrix}$$

Разность. Операция, во многом похожая на Пересечение, за исключением того, что в результирующем отношении содержатся кортежи, присутствующие в первом и отсутствующие во втором исходных отношениях. Обозначение: $\mathrm{R}_1 - \mathrm{R}_2$. Пример:

$$\sigma_{\mathrm{Q,T}}(\mathrm{R}) - \mathrm{S} = \begin{bmatrix} 5 & a \\ 3 & a \\ 9 & a \\ 1 & b \\ 2 & b \\ 4 & b \end{bmatrix} - \begin{bmatrix} 5 & a \\ 10 & b \\ 15 & c \\ 2 & d \\ 6 & a \\ 1 & b \end{bmatrix} = \begin{bmatrix} 3 & a \\ 9 & a \\ 2 & b \\ 4 & b \end{bmatrix}$$

Декартово произведение. Входные отношения могут быть определены по разным схемам. Схема результирующего отношения включает все атрибуты исходных. Кроме того:

– степень результирующего отношения равна сумме степеней исходных отношений
– мощность результирующего отношения равна произведению мощностей исходных отношений. Обозначение: $\mathrm{R}_1 \times \mathrm{R}_2$.Пример:

$$\sigma_{\mathrm{M,T}}(\mathrm{R}) \times (\sigma_{\mathrm{Q,T}}(\mathrm{R}) \cap \mathrm{S}) = \begin{bmatrix} x & a \\ y & a \\ z & a \\ w & b \end{bmatrix} \times \begin{bmatrix} 5 & a \\ 1 & b \end{bmatrix} = \begin{bmatrix} x & a & 5 & a \\ x & a & 1 & b \\ y & a & 5 & a \\ y & a & 1 & b \\ z & a & 5 & a \\ z & a & 1 & b \\ w & b & 5 & a \\ w & b & 1 & b \end{bmatrix}$$

Соединение /JOIN/. Эта операция имеет сходство с Декартовым произведением. Но здесь добавлено условие, согласно которому, вместо полного произведения всех строк, в результирующее отношение включаются только строки, удовлетворяющие определенному условию θ между атрибутами соединяемых отношений. Обозначение: $\pi_A(\sigma_\theta(R \times S))$. Пример:

$$\pi_{D1,D2,D3,D5}(\sigma_{D3=D4}(P \times Q)) = \begin{bmatrix} 1 & 11 & x & 1 \\ 1 & 11 & x & 2 \\ 2 & 11 & y & 1 \\ 4 & 12 & x & 1 \\ 4 & 12 & x & 2 \end{bmatrix}$$

Свойства параллельных систем [4]

Идеальная параллельная система обладает двумя главными свойствами: *линейное ускорение* (вдвое большее аппаратное обеспечение выполнит ту же задачу в два раза быстрее) и *линейная расширяемость* (вдвое большее аппаратное обеспечение выполнит вдвое большую задачу за то же время).

Более формально, если одна и та же работа выполняется на меньшей и на в N раз большей системе, то увеличение быстродействия (ускорение R), даваемое большей системой, определится как

$$R = T_1 / T_2.$$

Здесь T_1 – время, затраченное меньшей системой, T_2 – время, затраченное большей системой. Для линейного ускорения R = N. Ускорение позволяет определить эффективность наращивания системы на сопоставимых задачах.

Коэффициент расширяемости M определяется как

$$M = T_1' / T_2',$$

где T_1' – время, затраченное меньшей системой на решение небольшой задачи; T_2' – время, затраченное в N раз большей системой на решение в N раз большей задачи. Если M = 1, то расширяемость называется линейной. Расширяемость позволяет оценить эффективность наращивания системы на больших задачах.

Существуют два различных вида расширяемости: *пакетная* и *транзакционная*.

Если суть работы состоит в выполнении большого количества небольших независимых запросов от многих пользователей к базе данных коллективного пользования, то свойство расширяемости состоит в удовлетворении в N раз большего числа запросов от большего в N раз числа клиентов к большей в N раз базе данных. Этот вид расширяемости называется *транзакционным*. Он идеально подходит для параллельных систем, так как каждая транзакция представляет собой небольшую независимую работу, которая может выполняться на отдельном процессоре.

Пакетная расширяемость имеет место, когда задача состоит в выполнении одной большой работы. Она характерна для запросов к базам данных и для задач математического моделирования. В этих случаях расширяемость состоит в

использовании в N раз большего компьютера для решения в N раз большей задачи. Для систем баз данных пакетная расширяемость выражается во времени выполнения того же запроса к в N раз более крупной базе данных. Для научных задач – во времени выполнения того же расчета на в N раз более мелкой сетке или для в N раз более трудоемкого моделирования.

Достижимость линейного ускорения и линейного расширения затруднена тремя факторами:

- *Запуск* – время, необходимое для запуска параллельной операции. Если нужно запустить тысячи процессоров, то реальное время вычислений может оказаться значительно меньше времени, требуемого для их запуска.
- *Помехи* – появление каждого нового процесса ведет к замедлению всех остальных процессов, использующих те же ресурсы.
- *Перекос* – с увеличением числа параллельных шагов средняя продолжительность выполнения каждого шага уменьшается, но отклонение от среднего значения может значительно превзойти само среднее значение. Время выполнения работы – это время выполнения наиболее медленного шага работы. Когда отклонение от средней продолжительности превосходит ее саму, то параллелизм позволяет только слегка убыстрить выполнение работы.

Аппаратная архитектура систем баз данных [4 – 7]

Каким образом должна быть построена расширяемая многопроцессорная система баз данных? Стоунбрейкер [4] предложил следующую классификацию для целого спектра разработок (рис. 3 – 5).

Совместно используемая память (рис. 3). Все процессоры имеют прямой доступ к общей глобальной памяти и ко всем дискам. Примерами подобных систем являются мультипроцессоры IBM/370, Digital VAX, Sequent Symmetry.

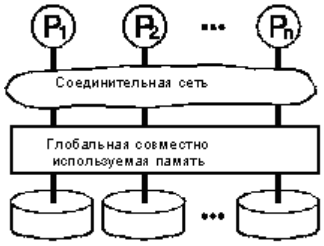

Рис. 3

Совместно используемые диски (рис. 4*)*. Каждый процессор имеет не только свою собственную память, но и прямой доступ ко всем дискам. Примерами являются IBM Sysplex и первоначальная версия Digital VAXcluster.

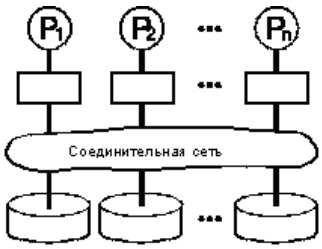

Рис. 4

Отсутствие совместного использования ресурсов (рис. 5). Каждая память и диск находятся в распоряжении какого-либо процессора, который работает как сервер хранящихся в них данных. Массовое запоминающее устройство в таких архитектурах распределено между процессорами посредством соединения одного или более дисков. Примерами таких машин являются Teradata, Tandem и nCUBE.

Рис. 5

Архитектуры без совместного использования ресурсов *сводят* к минимуму помехи посредством минимизации совместно используемых ресурсов. Кроме того, при использовании массово производимых процессоров и памяти им не требуется сверхмощная соединительная сеть. В таких архитектурах через сеть передаются только вопросы и ответы, а не большие массивы данных. Непосредственные обращения к памяти и к дискам обрабатываются локальным процессором, и только отфильтрованные (урезанные) данные передаются запрашивающей программе. Это позволяет реализовать более расширяемую архитектуру за счет минимизации трафика в соединительной сети.

Отсутствие совместного использования ресурсов характерно для систем баз данных, используемых в проектах Teradata, Gamma, Tandem, Bubba, Arbre и nCUBE. Системы для рабочих групп от 3com, Borland, Digital, HP, Novell, Microsoft и Sun также базируются на архитектуре типа клиент-сервер без совместного использования ресурсов.

Реальные соединительные сети, используемые в этих системах, зачастую совершенно не схожи друг с другом. Teradata использует избыточную древовидную соединительную сеть. Tandem – трехуровневую дуплексную сеть: два уровня внутри кластера и кольца, соединяющие кластеры. Единственное требование, которое Arbre, Bubba и Gamma предъявляют к соединительной сети, состоит в существовании связи между любыми двумя узлами. Gamma работает на Intel Hypercube. Прототип Arbre был реализован на основе процессоров IBM 4381, соединенных друг с другом в сеть напрямую. Системы для рабочих групп переходят с Ethernet на более высокоскоростные локальные сети.

Основным преимуществом мультипроцессоров без совместного использования ресурсов является то, что число процессоров в них может доходить до сотен и даже тысяч без возникновения каких-либо помех в работе одного со стороны другого. Компании Teradata, Tandem и Intel запустили проекты систем с более чем 200 процессорами. Intel разрабатывает гиперкуб с 2000 узлами. Максимальное число процессоров в многопроцессорной системе с разделением памяти равно к настоящему моменту 32.

Архитектуры без совместного использования ресурсов позволяют достичь почти линейного ускорения и расширяемости на сложных реляционных запросах и при транзакционной обработке запросов.

Вот почему проектировщики машин баз данных не видят смысла в выполнении сложных аппаратных и программных проектов с совместным использованием памяти и дисков.

Системы с совместным использованием памяти и дисков не так-то легко наращивать для приложений, связанных с базами данных. Основная проблема для мультипроцессоров с совместным использованием памяти – помехи. Соединительная сеть должна иметь пропускную способность, равную сумме пропускных способностей процессоров и дисков. Создать сеть, наращиваемую до тысячи узлов, – весьма непростая задача.

Для того, чтобы уменьшить трафик в сети и свести к минимуму время ожидания, каждому процессору придается большая собственная кэш-память. Эксперименты на мультипроцессорах с совместным использованием памяти, выполняющих задачи с базами данных, показывают, что загрузка и выталкивание из кэш-памяти значительно снижают производительность системы. При возрастании параллелизма помехи при совместном использовании ресурсов ограничивают рост производительности.

Для уменьшения помех в многопроцессорных системах часто используется механизм «родственного» планирования, предполагающий закрепление каждого процесса за конкретным процессором, что является формой разделения данных. Другие процессоры, желающие получить доступ к данным, посылают сообщения к серверам, управляющим этими данными. В системах с совместно используемой памятью разделение данных создает множество проблем, связанных с перекосом и распределением нагрузки. Поэтому при работе с базами данных такие мультипроцессоры могут быть экономично расширены только до нескольких процессоров.

Для борьбы с помехами в них часто применяется архитектура с совместным использованием дисков, что является логическим следствием родственного планирования. Если дисковая соединительная сеть наращивается до тысяч дисков и процессоров, то схема с совместным использованием дисков пригодна только для больших баз данных, предназначенных лишь для чтения, и для баз данных без одновременного использования. Эта схема мало эффективна для прикладных программ баз данных, которые считывают и записывают совместно используемые данные.

Если процессору нужно изменить какие-либо данные, он сначала должен получить текущую их копию. Так как другие процессоры могут в это время изменять те же самые данные, то процессор должен заявить о своих намерениях. Он может прочитать совместно используемые данные с диска и изменить их только в случае, если его намерение одобрено всеми остальными процессорами. Тогда они смогут учесть проведенные изменения в своей дальнейшей работе. Имеется множество вариантов оптимизации этого протокола. Но все они сводятся

к обмену сообщениями о резервировании данных и обмену большими физическими массивами данных. Это приводит к помехам, задержкам и большому трафику в совместно используемой соединительной сети.

Для прикладных программ с совместно используемыми данными подход с совместным использованием дисков обходится значительно дороже, чем подход без совместного использования ресурсов с обменом логическими вопросами и ответами высокого уровня между клиентами и серверами.

Такое решение возникло на основе применения мониторов обработки транзакций, которые разделяют нагрузку между раздельными серверами. Кроме того, оно основано на механизме вызова удаленных процедур.

Подчеркнем еще раз, что тенденция к разделению данных и к архитектуре без совместного использования ресурсов позволяет уменьшить помехи в системах с совместно используемыми дисками. Поскольку соединительную сеть системы с совместным использованием дисков практически невозможно расширить до тысяч процессоров и дисков, многие сходятся на том, что лучше с самого начала ориентироваться на архитектуру без совместного использования ресурсов.

Выработано единое мнение об архитектуре распределенных и параллельных систем баз данных. Эта архитектура базируется на идее аппаратного обеспечения без совместного использования ресурсов, когда процессоры поддерживают связь друг с другом только посредством передачи сообщений через соединяющую их сеть. В таких системах кортежи каждого отношения в базе данных разделяются между дисковыми запоминающими устройствами, напрямую подсоединенными к каждому процессору. Разделение позволяет нескольким процессорам просматривать большие отношения параллельно, не прибегая к использованию каких-либо экзотических устройств ввода/вывода.

Такая архитектура впервые была представлена компанией Teradata в конце 70-х годов и применена в нескольких исследовательских проектах. Теперь она используется в продуктах Teradata, Tandem, Oracle-nCUBE и еще нескольких продуктах, находящихся в стадии разработки. Исследовательское сообщество использовало архитектуру без совместного использования ресурсов в таких системах, как Arbre, Bubba и Gamma.

СУБД MYSQL CLUSTER

Целью этого занятия является знакомство с архитектурой, принципами работы,обеспечением надежности и основными управляющими командами СУБД MySQL Cluster.

Архитектура

Общее описание. MySQL Cluster – это параллельная СУБД, которая позволяет кластеризовать базу данных, хранящуюся в памяти системы без разделения ресурсов. Такая система позволяет работать с недорогим оборудованием при минимальных требованиях к аппаратуре и программному обеспечению. MySQL Cluster включает в себя стандартный сервер MySQL и механизм хранения данных, названный NDB, или NDB Cluster.

“Платформой” системы является набор компьютеров. На каждом из них запущены один или более процессов, которые могут включать в себя сервер MySQL, узел данных, управляющий сервер и, возможно, специализированные программы доступа к данным. Взаимосвязи между этими компонентами показаны на рис. 6.

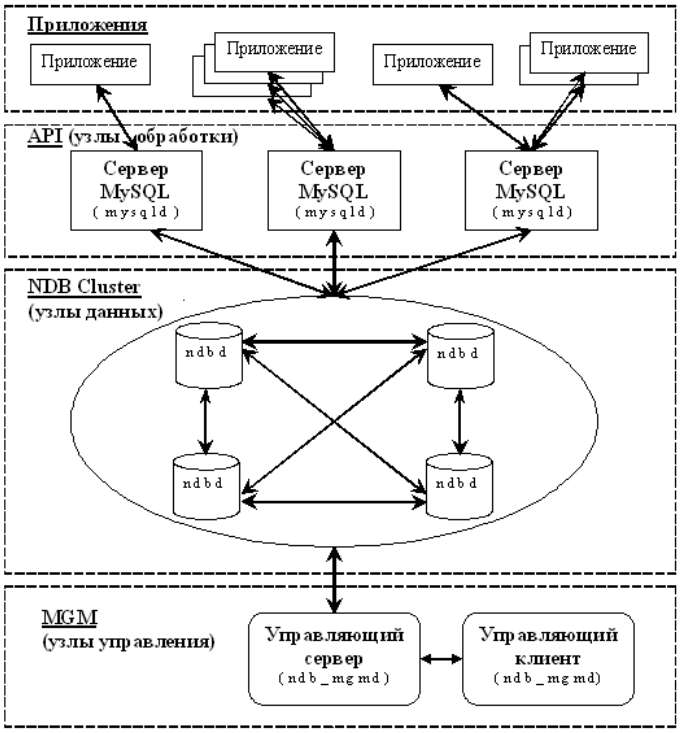

Рис. 6

MySQL Cluster состоит из трех типов узлов:

- *Управляющий узел (MGM)*. Функции этого типа узла – управление другими узлами в MySQL Cluster, предоставление конфигурационных данных, запуск и остановка узлов, запуск резервирования и т.д. Так как узел типа *MGM* управляет конфигурацией, то управляющие узлы должны быть запущены в первую очередь. *MGM* запускается при помощи команды ndb_mgmd.

- *Узел данных (NDB)*. Этот тип узлов хранит данные. Необходимое количество таких узлов зависит от числа копий базы данных и от количества фрагментов, на которые необходимо разбить базу. Узлы *NDB* брабатывают все транзакции.
- *Узел обработки (API)*. Узлы типа *API* имеют непосредственный доступ к данным, хранимым в NDB Cluster. Это обычные сервера MySQL, настроенные на использование NDB Cluster. MySQL сервер предоставляет стандартный SQL интерфейс. Сервер обрабатывает запросы, преобразуя их транзакции к узлам данных.

Принципы работы

Конфигурирование кластера включает конфигурирование каждого индивидуального узла в кластере и установление индивидуальных коммуникационных связей между узлами. MySQL Cluster разработан, исходя из предположения, что все узла хранения примерно одинаковы по вычислительной мощности, размерам памяти и полосы пропускания.

MySQL Cluster держит все данные в оперативной памяти для транзакций и быстрого восстановления, ограничивая тем самым операции ввода/вывода. Запись на диск производится асинхронно для протоколирования транзакции.

Данные распределены следующим образом. Все таблицы поделены горизонтально на разделы. Каждому разделу отвечает свой узел хранения. Все узлы могут быть поделены на группы. Между узлами одной группы данные зеркалируются. Это делается для повышения надежности системы (вопрос будет рассмотрен далее). Число групп зависит от числа копий базы данных и от общего количества узлов хранения. Например, если имеются 6 узлов хранения и 2 копии базы данных, то число групп равно 3 по два узла хранения в каждой.

Параллельная обработка в MySQL Cluster использует два вида параллелизма:

- межзапросный параллелизм;
- внутризапросный параллелизм.

Межзапросный параллелизм означает выполнение множества запросов на независимых API узлах. Его реализация затруднена тем, что MySQL Cluster не содержит утилит для распределения нагрузки между независимыми API.

Внутризапросный параллелизм подразумевает разделение данных между узлами хранения данных. Как отмечено ранее, каждый узел хранения содержит в оперативной памяти свой раздел базы данных. Для API узлов, выполняющих основную обработку, узлы данных представляют собой кэш базы данных.

Параллельная архитектура MySQL Cluster позволяет достичь, по словам разработчиков, почти линейной расширяемости. Предположительно, линейная расширяемость может быть достигнута при одновременном увеличении числа узлов хранения и числа API узлов. Но здесь все не так просто, ибо с ростом числа узлов возрастает влияние «накладных расходов» (коллизии в сети, латентность, неравномерное распределение нагрузок по процессорам и др.). Это может

существенно влиять на производительность системы. Так что вопрос о масштабируемости MySQL Cluster пока остается открытым.

MySQL Cluster поддерживает следующие сетевые протоколы: TCP/IP и SCI. TCP/IP используется по умолчанию для установления соединения между узлами. Соединения устанавливаются автоматически, исходя из конфигурации кластера. При этом предусмотрена возможность самостоятельной настройки сетевых взаимодействий в кластере.

Обеспечение надежности в MySQL Cluster

Архитектура MySQL Cluster была спроектирована для достижения высокой надежности сервера MySQL. С этой целью:

- API узлы подключены ко всем узлам хранения. Если в одном из узлов хранения произойдет сбой, API узел может использовать другой узел для выполнения транзакции;
- хранимая информация зеркалируется. Если какой-либо узел хранения «сбоит», то всегда найдется другой узел с такой же информацией;
- управляющий узел может быть остановлен без какого-либо влияния на остальные узлы кластера.

При помощи такого подхода к проектированию удалось исключить сбой всей системы из-за сбоя одного узла. Любой узел может быть остановлен без влияния на работу систему в целом.

Рассмотрим этот подход более подробно.

Синхронное зеркалирование. Все данные базы данных зеркалируются в пределах узлов хранения одной группы. Число копий баз данных может варьироваться от 1 до 4. Оно выбирается администратором при запуске кластера. Зеркалирование происходит во время выполнения транзакции. Синхронное зеркалирование позволяет при сбое одного из узлов заменить его менее чем за одну секунду. Если во время выполнения транзакции произошел сбой и не выполнено зеркалирование, приложение уведомляется об этом. Так что всегда есть возможность повторения сбойной транзакции.

Обнаружение сбоя. Различают два способа определения сбойного узла – т.н. «потеря связи» и «потеря пульса». В обоих случаях посылается сообщение всем узлам хранения и определяется дальнейшая возможность функционирования системы.

При сбое возможна ситуация, когда кластер будет поделен на несколько независимо работающих систем. Ее возникновение влечет потерю достоверности и целостности данных в кластере. Для устранения подобных ситуаций используется специальный сетевой протокол разбиений. Согласно ему выбирается одна из работоспособных частей, а остальные узлы перезапускаются и присоединяются к системе как новые.

Потеря связи. Узлы соединятся через различные протоколы. При нормальной работе MySQL Cluster все узлы хранения соединены друг с другом и каждый узел API соединен со всеми узлами хранения. Если узел хранения сообщает, что связь между двумя узлами потеряна, то об этом незамедлительно информируются другие узлы и все вместе они находят сбойный узел. Все сбойные узлы автоматически перезапускаются и подсоединятся к MySQL Cluster. Потеря связи – это наиболее быстрый способ нахождения сбойного узла.

Потеря "пульса". Существуют сбои узлов, которые невозможно определить при помощи потери связи, такие как проблемы с дисками, оперативной памятью, «истощением» процессора и т.д. Эти сбои влияют на корректную работу узла, но не влияют на связь узла с оставшейся частью кластера. Узлы хранения организованы в логический цикл. Каждый узел посылает сигналы "пульса" следующему узлу хранения в цикле. Узел должен послать последовательно 3 сигнала "пульса" (рис. 7). Если он этого не сделал, то следующий в цикле узел хранения классифицирует его как сбойный.

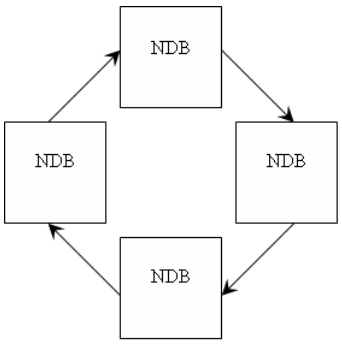

Рис. 7

Восстановление системы. В MySQL Cluster используются следующие приемы для восстановления системы после сбоя.

Протоколирование. Во время "нормальных" операций на диск асинхронно пишется протокол операций с базой данных (insert, delete, update и т.д.). Протоколы пишутся на диск каждого узла хранения и используются только при сбое всей системы. Так как протокол содержит все операции изменения, то легко восстановить состояние базы данных на момент сбоя.

Локальные точки восстановления. С ростом числа операций увеличивается и объем протокола. Чтобы он не разрастался, создаются локальные точки восстановления – сохранение базы данных на диск в определенные моменты времени. При сбое отдельного узла хранения восстанавливается сохраненная база, и к ней применяется восстановление по протоколу. Локальные точки восстановления создаются независимо на каждом узле хранения.

Глобальные точки восстановления. Для восстановления после сбоя всей системы создаются глобальные точки восстановления. Одновременно с их формированием происходит формирование локальных точек восстановления на всех узлах хранения.

Процедуры конфигурирования MySQL Cluster

Настройка MGM node выполняется в порядке:

1. Создать каталог /var/lib/mysql-cluster.
2. В каталоге /var/lib/mysql-cluster/ создать файл config.ini:

```
#секция настроек NDB node по умолчанию
[ndbd default]
#количество копий (1+количество резервных копий)
NoOfReplicas=2
#количество оперативной памяти, отводимое под данные
DataMemory=1600M
#количество оперативной памяти, отводимое под индексы
IndexMemory=250M
#место хранения данных на узлах хранения
DataDir=/var/lib/mysql-cluster/db
#[ndb_mgmd] секция отвечает за параметры MGM node
[ndb_mgmd]
#IP адрес MGM node
HostName=192.168.214.40
[ndb_mgmd]
#IP адрес MGM node
HostName=192.168.214.42
#секция [ndbd] отвечает за параметры NDB node
[ndbd]
#IP адрес NDB node
HostName=192.168.214.41
[ndbd]
#IP адрес NDB node
HostName=192.168.214.45
#секция  отвечает за параметры API node
[mysqld]
#IP адрес API node
HostName=192.168.214.40
 [mysqld]
#IP адрес SQL node
HostName=192.168.214.42
#настройка стека tcp
[tcp default]
#размер буфера посылаемых данных
```

SendBufferMemory=2M

#размер буфера принимаемых данных

ReceiveBufferMemory=1M

3. Скопировать из каталога /usr/bin в каталог */var/lib//mysql-cluster файл ndb_mgmd*.

Настройка NDB node :

1. Создать каталог /var/lib/mysql-cluster.
2. Скопировать из /usr/bin в /var/lib/mysql-cluster файл ndbd.

Настройка API node :

1. В каталоге /etc создать файл my.cnf.

#настройки, используемые программой mysqlclient

[client]

#порт для подключения к API node

port = 3306

#UNIX сокет для подключения к API node

socket= /tmp/mysql.sock

#настройки для MySQL server

[mysqld]

#порт для подключения к MySQL server

port = 3306

#UNIX сокет для подключения к MySQL server

socket= /tmp/mysql.sock

skip-locking

key_buffer = 256M

max_allowed_packet = 1M

table_cache = 256

sort_buffer_size = 1M

read_buffer_size = 1M

read_rnd_buffer_size = 4M

myisam_sort_buffer_size = 64M

thread_cache_size = 8

query_cache_size= 16M

thread_concurrency = 8

#настройки для использования MySQL Cluster

#место хранения схем таблиц

datadir=/var/lib/mysql

#подключение NDB nodes

ndbcluster
#адреса MGM nodes
ndb-connectstring = 192.168.214.40, 192.168.214.42
[mysql-cluster]
Ndbcluster
#адреса MGM nodes
ndb-connectstring = 192.168.214.40, 192.168.214.42
[mysqldump]
quick
max_allowed_packet = 16M
[mysql]
no-auto-rehash
[isamchk]
key_buffer = 128M
sort_buffer_size = 128M
read_buffer = 2M
write_buffer = 2M
[myisamchk]
key_buffer = 128M
sort_buffer_size = 128M
read_buffer = 2M
write_buffer = 2M
[mysqlhotcopy]
interactive-timeout

2. Создать каталог /var/lib/mysql. В нем будут храниться системные базы данных и схемы баз данных, используемых MySQL Cluster.
3. Создать группу mysql:

shell> groupadd mysql

4. Создать пользователя mysql:

shell>useradd -g mysql mysql

5. Перейти в каталог /usr/local/mysql:

shell>cd /usr/local/mysql

6. Изменить владельца каталога:

shell> chown -R mysql

7. Изменить группу-владельца каталога:

shell> chgrp -R mysql

8. Создать системные таблицы SQL node:

/usr/bin/mysql_install_db –-user=mysql

9. Изменить владельца каталога:

shell> chown -R root

10. Изменить владельца каталога данных:

shell> chown -R mysql <каталог_баз_данных>

Управление MySQL Cluster

Консоль управления ndb_mgm. В комплект поставки MySQL Cluster входит бинарный файл ndb_mgm. После компиляции он находится в каталоге /usr/local/bin. Запуск производится командой /usr/local/bin/ndb_mgm. Файл может быть запущен на любом узле MySQL Cluster. Ниже приведено описание команд, используемых для управления кластером:

1. HELP – выводит список доступных команд.
2. SHOW – выводит состояние MySQL Cluster.
3. CONNECT ip_адрес_MGM_node – подключение к MGM node.
4. номер_узла START – запуск узла хранения с номером номер_узла. Для работы команды необходимо запустить узел хранения с опцией -n или –nostart.
5. номер_узла STOP – останов узла хранения с номером номер_узла.
6. номер_узла RESTART – перезапуск узла хранения с номером номер_узла.
7. номер_узла STATUS – вывод информацию о состоянии узла.
8. ENTER SINGLE USER MODE номер_узла – однопользовательский режим, в котором доступ к MySQL Cluster имеет доступ только SQL node с номером номер_узла.
9. EXIT SINGLE USER MODE – выход из однопользовательского режима.
10. SHUTDOWN – останов кластера, за исключением SQL nodes.
11. QUIT, EXIT – выход из управлающей консоли.

Порядок запуска MySQL Cluster:

1. На каждом MGM node выполнить команду:

shell>/var/lib/mysql-cluster/ndb_mgmd --config-file=/etc/config.ini

2. На каждом NDB node выполнить команду:

shell>/var/lib/mysql-cluster/ndbd --connect-string = ”<список_узлов_управления>” [--initial]

где <список_узлов_управления> – IP адреса MGM nodes, написанные через запятую. Если требуется очистить память NDB node от данных, то необходим добавить ключ --initial.

3. На каждом SQL node выполнить команду:

shell>*/usr/bin/mysqld_safe &*

Порядок останова кластера. Для штатного останова используется управляющая консоль:

ndb_mgm>connect <ip_адрес_узла_управления>
ndb_mgm>show
ndb_mgm>shutdown

SQL node останавливаются при необходимости вручную:

shell>ps ax | grep mysql

Будет выведен список процессов. Первым необходимо останавливать процесс, содержащий *mysqld_safe*.Вторым–процесс *mysqld*.

Вопросы для самоконтроля:

1. Что такое MySQL Cluster?
2. Какой тип архитектуры использует MySQL Cluster?
3. Механизм, используемый MySQL Cluster для хранения данных
4. Из каких типов узлов состоит MySQL Cluster?
5. Каким образом распределены данные между узлами хранения?
6. Какие виды параллелизма использует MySQL Cluster?
7. Какие сетевые протоколы использует MySQL Cluster?
8. Назовите технологии, использующиеся в MySQL Cluster для обеспечения надежности.

ПАРАЛЛЕЛЬНАЯ СУБД CLUSTERIX

Целью этого занятия является ознакомление с принципами построения параллельной СУБД Clusterix, получение навыков работы на кластере баз данных.

Общее описание. Рассматриваемая СУБД ориентирована на использование стандартного общедоступного аппаратно-программного обеспечения (Beowulf-технология). Аппаратно СУБД Clusterix функционирует на кластере из персональных компьютерных (ПК), соединенных локальной сетью FastEthernet или GigabitEthernet. Программная среда параллельной СУБД Clusterix представляет собой “надстройку” над серверами СУБД MySQL, установленными на всех узлах кластера. В функции СУБД MySQL входит реализация таких низкоуровневых операций с БД, как хранение и обработка отношений БД в виде специализированных файлов.

Функции программной надстройки заключаются:

- в организации параллельной обработки запроса пользователя различными узлами кластера;
- в управлении и мониторинге за состоянием узлов кластера;
- в преобразовании исходного SQL-запроса пользователя в соответствующий план обработки, который обеспечивает параллельное его исполнение.

Взаимодействие между надстройкой и серверами СУБД MySQL осуществляется передачей SQL-запросов (команды) и через файлую подсистему (данные).

Программное обеспечение кластера функционирует под операционной системой Linux.

В основу параллельной обработки положена идея конвейерной обработки запроса [10]. Эта идея реализуется построением специального план обработки запроса (рис. 8) по схеме:

“SELECT – PROJECT - JOIN”.

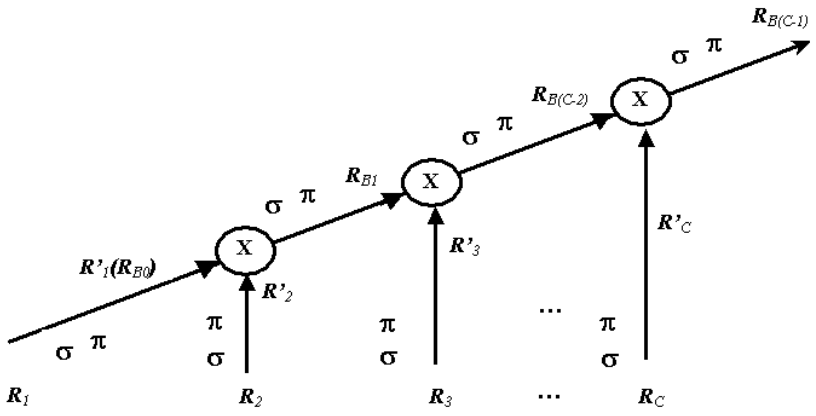

Рис. 8

На рис. 8: $\mathbf{R_i}$ – одно из отношений, участвующих в обработке запроса, i=1…C. Здесь C – количество отношений, задействованных в запросе; R'_i –

промежуточное отношение, i=1…C; $\mathbf{R_{Bj}}$ – временное отношение, j=1…C – 1; **σ** – операция селекции; **π** – операция проекции; **x** – операция декартова произведения.

Обработка запроса по указанной схеме проводится в 3 этапа.

На первом этапе выполняется выборка кортежей из исходного отношения БД (R_i), отфильтрованных условием *селекции*(**σ**). На втором этапе над этими кортежами выполняется операция *проекции* (**π**) только тех полей (атрибутов) отношения, которые будут участвовать в последующей обработке или должны присутствовать в конечном результате обработки запроса. Назначением операций **σ** и **π** является уменьшение размеров исходного отношения БД перед третьим этапом – операцией *соединения*. Выполнение операции соединения является одной из наиболее трудоемких операций. Ее эффективность напрямую зависит от размеров отношений, участвующих в соединении.

Система предусматривает двухуровневую обработку данных: верхний и нижний уровни обработки.

Нижний уровень выполняет операции селекции (**σ**) и проекции (**π**) над исходными отношениями $\mathbf{R_i}$ базы данных. Результатом обработки нижнего уровня является промежуточное отношение $\mathbf{R'_i}$. Далее промежуточное отношение передается на верхний уровень для дальнейшей обработки – операции соединения.

На *верхнем уровне* выполняется операция соединения между отношениями $\mathbf{R'_i}$ и $\mathbf{R_{B(i-2)}}$. Результатом выполнения соединения на i-шаге является временное отношение $\mathbf{R_{B(i-1)}}$.

Операция соединения

$$R \blacktriangleright\blacktriangleleft S = \pi\,(\sigma_\theta(RxS)).$$

выражается через операции селекции, проекции и декартова произведения.

Отношения БД распределены по дискам на процессорах нижнего уровня (IO_r). Распределение отношений осуществляется горизонтально с применением хеш-функции к первичному ключу для каждого кортежа отношения.

Процессоры верхнего уровня обработки запроса называются процессорами JOIN. Количество процессоров JOIN равно количеству процессоров *IO*.

Кроме исполнительных процессоров (IO и JOIN), есть еще два процессора. Процессор MONITOR реализует функции мониторинга и управления остальными процессорами системы. Процессор (препроцессор) MTRANS предназначен для претрансляции исходного запроса пользователя к виду регулярного дерева обработки запроса (рис. 8). Оба они функционируют на *Host* ЭВМ.

Принятый план обработки ориентирован на реализацию следующих принципов:

– параллелизм;

– конвейерность;

– регулярность процедуры сборки промежуточных отношений для выполнения страничного соединения;

– реализация промежуточного хеширования для повышения эффективности страничных соединений и уменьшения объемов сетевых передач.

Оператор SELECT осуществляет выборку кортежей из отношений БД, PROJECT – проекцию полей отношения, JOIN выполняет соединение. Операторы SELECT и PROJECT относятся к первому, JOIN – ко второму уровню обработки данных в кластере. Операторы первого уровня выделены в отдельный программный модуль – процессор ввода/вывода (IO), операции второго уровня – в процессор JOIN.

Основные программные модули («логические» процессоры) разрабатываемой системы:

- Модуль мониторинга и управления **mlisten** (MONITOR) – получение SQL запросов пользователя, передача результатов обработки, управление компонентами системы.
- Модуль ввода/вывода **irun** (IO) – выполнение операций селекции, проекции, ввода/вывода.
- Модуль соединения **jrun** (JOIN) – выполнение операций сортировки, соединения, проекции над промежуточными результатами обработки.
- Модуль сортировки и агрегации **msort** (SORT) – выполнение операций агрегации, группировки и сортировки над результатами обработки.
- Модуль трансляции **mtrans** (MTRANS) – выполнение трансляции SQL запросов во внутреннее представление системы.

На каждом физическом процессоре любого уровня могут быть запущены несколько «логических». Число «логических» процессоров на обоих уровнях всегда одинаково, а отношение k = q/p необходимо целое, k = 1, 2, … , где q – число физических процессоров JOIN, p – число физических процессоров IO.

Благодаря модульному построению можно получить несколько вариантов архитектуры системы: «линейка» (рис. 9), «симметрия» (рис. 10), «асимметрия» (рис. 11). Здесь кружками обозначены «логические» процессоры, прямоугольниками – физические процессоры.

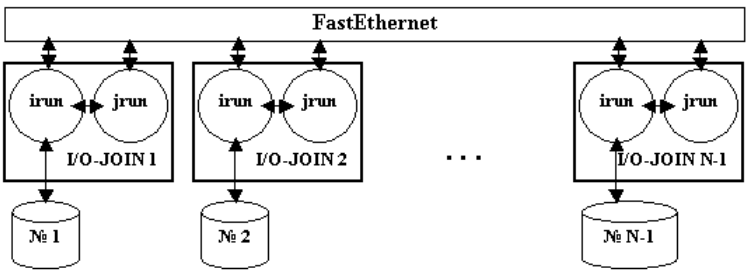

Рис. 9

Архитектура «линейка» имеет место при совмещении на одном физическом процессоре двух «логических» – IO и JOIN ($k \to \infty$).

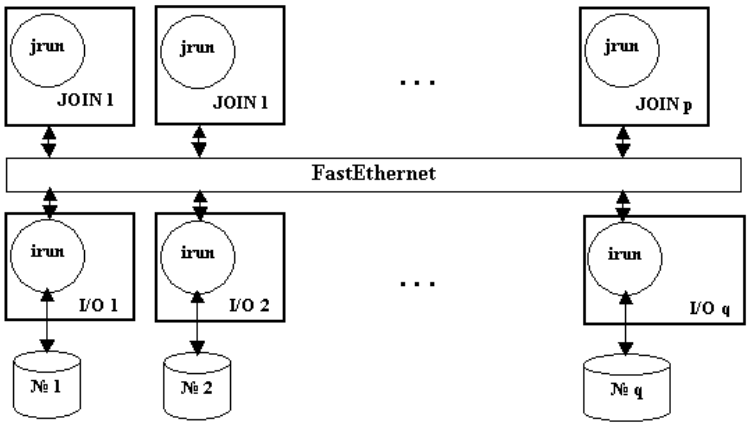

Рис. 10

Архитектура «симметрия» получается в том случае, если число процессоров верхнего уровня равно числу процессоров нижнего уровня. При этом на каждый «логический» процессор выделяется один физический процессор (k = 1).

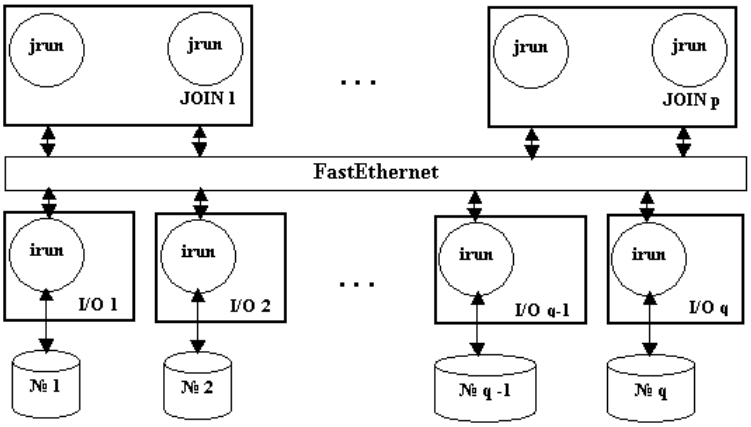

Рис. 11

Архитектура «асимметрия» имеет место, если число процессоров верхнего уровня (JOIN) не равно числу процессоров нижнего уровня (IO). Обычно здесь k = 2, 3, …, т.е. на одном «физическом» процессоре JOIN совмещаются два и более «логических».

Взаимодействие между модулями осуществляется передачей сообщений сетевыми протоколами TCP/IP и реализовано через «сокеты». Программные модули написаны с использованием технологий параллельного программирования (threads, mutex, shared memory). На каждом процессоре функционирует СУБД MySQL, которая выполняет низкоуровневые операции: работа с файлами, индексами и т.д.

Алгоритм работы системы. Точкой входа в систему для пользователя является Host-машина. Пользователь передает свой SQL-запрос с помощью

программы-клиента *sql_client*. Программа *sql_client* подключается к модулю *mlisten* на заранее известный сетевой порт и посылает SQL-запрос в виде строки. Модуль *mlisten* при подключении данного пользователя запускает для его обработки функцию-поток *Процесс h*, где *h* – порядковый номер запроса в системе. Получив строку запроса, *Процесс h* передает ее в подсистему трансляции, которая преобразует исходный запрос к плану обработки запроса.

План обработки данного запроса формируется в виде набора SQL-команд для каждого из трех программных модулей: *irun*, *jrun* и *msort*. Команды пересылаются соответствующим модулям, а сам *mlisten* переходит в режим ожидания результата обработки. На каждый запрос пользователя выделяется отдельный поток-обработчик (функция *Client* файла *mlisten.cpp*). Первым к обработке запросов приступает модуль *irun*. На рис. 12 поток управляющих команд показан пунктиром, поток данных – сплошной линией, сетевые интерфейсы – черными точками.

Модуль **irun** выполняет несколько потоков-обработчиков. Каждый обработчик функционально ориентирован. Получив пакет команд от *Процесс h*, модуль *irun* помещает его во входную очередь команд (обработчик *ServGet*). Другой обработчик – *Run* – занимается тем, что последовательно выбирает команды из входной очереди команд и с помощью API-функций отправляет их на выполнение СУБД MySQL. От СУБД результат обработки возвращается в виде буфера строк, каждая строка которого есть кортеж результата. Далее над этим буфером проводится операция динамического сегментирования по другим модулям *irun*.

В результате на каждом модуле *irun* формируется промежуточное отношение в виде трех файлов формата MySQL (*.frm – файл структуры отношения, *.MYD – файл хранения данных отношения, *.MYI – файл хранения индексов отношения). Полученное отношение передается соответствующему модулю *jrun*, а сам модуль *irun* приступает к выполнению следующего SQL-запроса из входного буфера команд. Это происходит до полного освобождения входного буфера.

Модуль **jrun** помещает полученный от *mlisten* пакет команд во входную очередь. После приема (обработчик *ServGet*) промежуточного отношения от *irun* модуль *jrun* помещает его в системный каталог СУБД MySQL (по умолчанию */var/lib/mysql/in_*). Из входного буфера извлекается текущая команда, которая передается на выполнение СУБД MySQL.

Команда состоит из трех операций. Первая операция создает индекс временного отношения по атрибуту (обработчик *Index*), который будет принимать участие в операции соединения на текущем шаге. Вторая – непосредственно операция соединения.Ее результатом является буфер строк, который затем сегментируется по другим модулям *jrun*. Операция динамического сегментирования осуществляется на основе атрибута, по которому будет производиться соединение на следующем шаге. В результате этих двух операций формируется временное отношение.

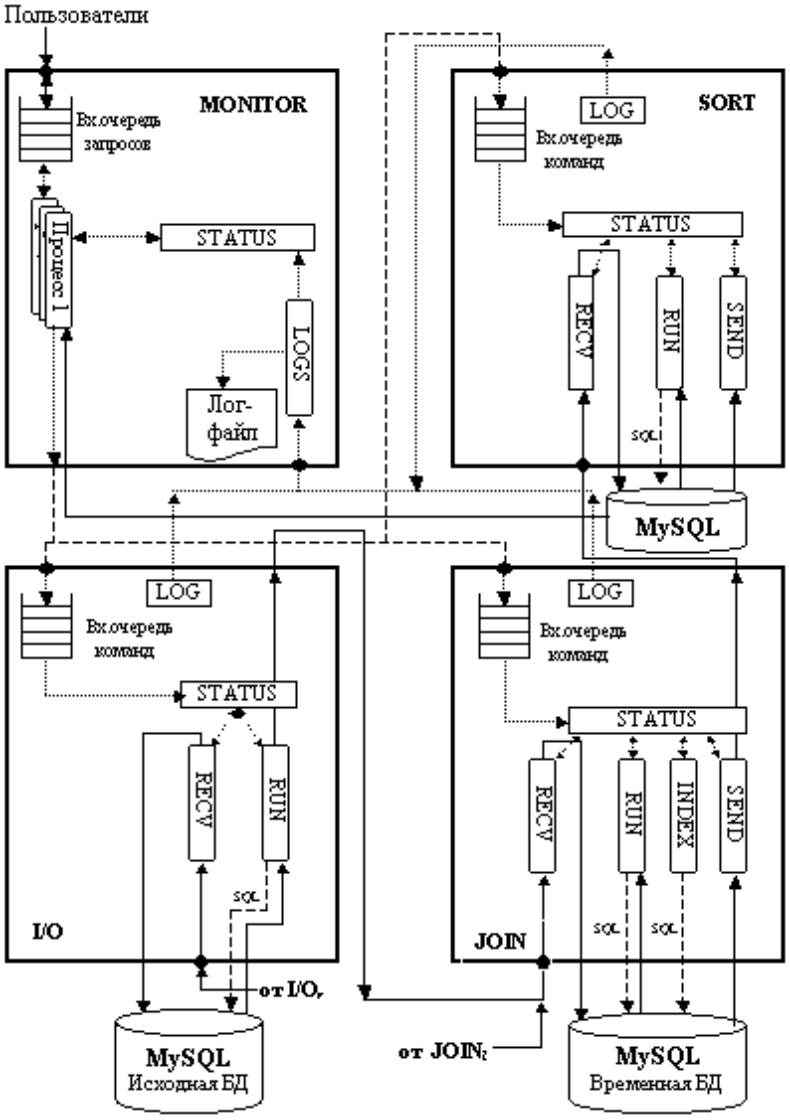

Рис. 12

И, наконец, последняя операция – создание индекса временного отношения по атрибуту, по которому проводилось сегментирование.

При достижении конца очереди команд для текущего запроса полученное временное отношение пересылается на следующий уровень обработки – модуль *msort*. Сам же модуль *jrun* приступает к обработке очередного запроса при наличии соответствующего временного отношения. Здесь инициирующим действием на запуск алгоритма обработки является получение промежуточного отношения от *irun*. Тем самым реализуется управление по готовности данных.

Модуль **msort** также имеет входную очередь команд. Каждый запрос состоит из одной команды. При получении временных отношений от всех модулей *jrun* модуль *msort* объединяет их содержимое в одно результирующее отношение. Далее к отношению применяется операция агрегации и сортировки, если они предусмотрены в исходном запросе пользователя. Модуль *mlisten* определяет факт завершения обработки запроса модулем *msort* и выходит из цикла ожидания. Полученное отношение передается пользователю в качестве результата.

На этом процесс обработки текущего запроса завершается.

Каждый модуль имеет в своем составе структуру STATUS (рис. 12). Эта структура выполняет несколько функций:

- сбор и хранение информации о прохождении запросом различных стадий обработки в рамках модуля;
- организация взаимодействия (локальной маршрутизации) между одновременно работающими обработчиками;
- обеспечение строгой последовательности обработки запроса.

Наличие двух уровней обработки с использованием СУБД MySQL обуславливает необходимость перемещения данных между уровнями обработки (IO, JOIN, SORT). Кроме хеширования отношений при распределении на страницы, в системе реализован механизм динамического сегментирования временных и промежуточных отношений на основе хеширования по полям, участвующим в операции соединения.

В системе реализован алгоритм операции соединения по равенству полей. Он не является универсальным. Поэтому для всех других вариантов соединения приходится организовывать “кольцо”, когда промежуточное отношение от модуля IO последовательно передается через все модули JOIN.

Хранение данных в системе. Данные СУБД *горизонтально* распределены по узлам IO. Распределение данных между узлами IO осуществляется с помощью хеширования значений первичного ключа. Для отношения, первичный ключ которого состоит из нескольких полей, функция хеширования

hash=(key_field1 mod M + … + keyfieldP mod M) mod M,

где P – количество полей в первичном ключе; основание деления по модулю, mod – операция деления по модулю. Значение *hash* показывает порядковый номер процессора IO, на который распределяется текущий кортеж.

Команды управления кластером. Для управления кластером используется программа-скрипт mgm_clusterix:

shell> mgm_clusterix имя_команды.

Основные функции скрипта:

- задание архитектуры и конфигурации кластера (команда *set_conf*);
- запуск (команда start) и останов (команда *stop*) кластера;
- формирование тестовой базы данных (команда *db*);
- проверка статуса кластера (команда *stat*);
- обновление версии программного обеспечения (команда conf);
- перезагрузка узлов кластера (команда *reboot*);
- выключение узлов кластера (команда *poweroff*);
- вызов справки по всем функциям системы (команда *help*).

Удаленное управление узлами кластера осуществляется с помощью службы *ssh*. Для этого на каждый узел установлен агент *ssh*, сформированы и распределены ключи. Прежде чем начать выполнение команд с помощью *mgm_clusterix*, необходимо выполнить следующие команды:

shell> ssh–agent $SHELL

shell> ssh–add

После этого система попросит ввести пароль для ключа. Операционная система будет хранить пароли в специальном системном буфере и подставлять их по мере необходимости. Если этого не сделать, система будет запрашивать пароль каждый раз при выполнении команды *ssh* на удаленном узле кластера.

Запуск системы осуществляется командой:

shell> *./mgm_clusterix start*

Результатом ее выполнения является список стартовавших модулей с их IP – адресами и номерами сетевых портов.

Останов системы реализуется командой:

shell> *./mgm_clusterix stop*

Результатом выполнения является список остановленных модулей с их IP – адресами и номерами сетевых портов.

Задание конфигурации. Конфигурация системы задается командой:

shell> *./mgm_clusterix set_conf номер_конфигурации*

Параметр *номер_конфигурации* представляет собой условное обозначение конфигурации системы. В табл. 1 представлены возможные значения этого параметра.

Таблица 1

Номер конфигурации	**Архитектура**	**Кол-во модулей I/O**	**Кол-во модулей JOIN**	**Кол-во узлов кластера**
422	симметрия	2	2	4
433	симметрия	3	3	6
444	симметрия	4	4	8
522	линейка	2	2	2
544	линейка	4	4	4
588	линейка	8	8	8
424	асимметрия	4	2	6
426	асимметрия	6	2	8

Проверка статуса системы осуществляется командой

shell> *./mgm_clusterix stat*

Результатом работы команды является список модулей с их IP-адресами, номерами сетевых портов и статусом выполнения («Running»– выполняется, «Stoped» – остановлен, «Not Found» – не определен).

Формирование тестовой базы данных. В качестве тестовой базы данных используется база данных теста TPC–D. Для каждой конфигурации формируется своя тестовая база данных. Ее формирование осуществляется командой

shell> *./mgm_clusterix db*

Эта команда включает в себя следующие этапы выполнения:

− хеширование отношений исходной базы данных по первичным ключам для каждого IO-узла;

− формирование фрагментированных отношений тестовой базы данных;

− распределение фрагментированных отношений тестовой базы данных по IO-узлам системы.

Обновление версии программного обеспечения. Эта функция реализуется командой:

shell> *./mgm_clusterix conf.*

Команда *conf* состоит из следующих операций:

− замена программных модулей на удаленных узлах кластера версией с сервера управления;

− удаление всех отношений из временной базы данных на удаленных узлах кластера.

Перезагрузка узлов кластера реализуется командой:

shell> ./mgm_clusterix reboot.

Результатом выполнения этой команды является перезагрузка операционной системы на всех узлах кластера.

Выключение узлов кластера реализуется командой:

shell> ./mgm_clusterix poweroff

Результатом выполнения этой команды является выключение питания узлов кластера.

Запуск кластера выполняется в следующем порядке:

1. задать текущую конфигурацию кластера;
2. распределить базу данных по множеству узлов кластера;
3. запустить кластер;
4. проверить текущее состояние кластера;
5. передать на выполнение SQL – запрос, зафиксировать результат обработки запроса;
6. остановить кластер.

Соответствующая последовательность команд:

- задание конфигурации кластера

shell> mgm_clusterix set_conf номер_конфигурации

- подготовка БД для текущей конфигурации

shell> mgm_clusterix db

- запуск кластера

shell> mgm_clusterix start

- проверка статуса кластера

shell> mgm_clusterix stat

- запуск SQL-запроса

shell> sql_client IP_адрес Port_номер SQL_команда

- остановка работы кластера

shell> mgm_clusterix stop

Вопросы для самоконтроля*:*

1. Каким образом должна быть построена расширяемая многопроцессорная система баз данных?
2. Опишите систему с совместно используемой памятью.
3. Опишите систему с совместно используемыми дисками.
4. Что означает отсутствие совместного использования ресурсов?
5. Приведите примеры систем баз данных, для которых характерно отсутствие совместного использования ресурсов.

ОБРАБОТКА ЗАПРОСОВ В СУБД CLUSTERIX

Назначение СУБД Clusterix – параллельное выполнение SQL-запросов за минимально возможное время. Для достижения этой цели исходный запрос пользователя преобразуется к виду, позволяющему выполнять части исходного запроса параллельно. Наиболее трудоемкими операциями СУБД являются сортировка и соединение.

Формирование команд плана обработки запросов

Пример построения дерева обработки. Рассмотрим конфигурацию, содержащую 2 узла IO и 2 узла JOIN. Пусть имеется база данных с тремя отношениями A, B, C:

Отношение A: A1* A2 A3
Отношение B: B1* B2
Отношение C: C1* C2* C3

* – атрибуты, входящие в первичный ключ отношения.

Построим дерево обработки следующего запроса:

```
SELECT A2, SUM(C3) FROM A, B, C
WHERE A1=C2 AND B1=C1 AND B2>50
GROUP BY A2
```

Для упрощения процесса построения дерева на начальных стадиях будем считать, что 1) имеется по одному узлу на каждый логический процессор обработки (IO, JOIN, SORT); 2) одни и те же команды выполняются параллельно на нескольких узлах обработки (IO, JOIN).

Сначала надо определить, сколько отношений присутствует в запросе. По части запроса за ключевым словом FROM убеждаемся в том, что в нашем случае таких отношений три – A, B, C. Поэтому схема плана обработки отвечает рис. 13.

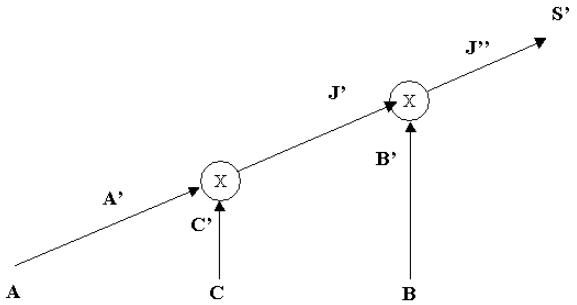

Рис. 13

Итак, общая схема обработки запроса известна. Остается определить конкретные команды этого плана.

Состав команд. Все планы обработки запроса Clusterix состоят из трех типов команд: для IO, для JOIN, для SORT. Формирование команд идет слева направо, по ходу выполнения запроса.

Так как обработка запроса начинается с IO, определим первую команду для IO – A`. В результате выполнения этой команды получается промежуточное отношение, содержащее все атрибуты отношения A, необходимые для дальнейшей обработки запроса.

Общий вид команды для IO выглядит следующим образом:

SELECT *атрибуты_отношения* FROM *имя_отношения* WHERE *условие_селекции*

Сначала определяется имя текущего отношения. В нашем случае – это отношение A.

Затем определяются *атрибуты_отношения.* Если атрибут присутствует в отношении A, но отсутствует в данном запросе, то его использовать не нужно. В этом состоит смысл операции *проекция* – исключение из обработки неиспользуемых атрибутов. Перечислим все атрибуты отношения A, участвующие в рассматриваемом запросе. Таких атрибутов два: A1 и A2. Атрибут A3 имеется в исходном отношении A, но в данном запросе не используется. Поэтому он не включается в команду.

Далее определяется *условие_селекции*. Оно задается условиями исходного запроса

<атрибут_отношения_A> условие <константа>,

которые располагаются после ключевого слова WHERE. Для отношения A таких условий в рассматриваемом запросе нет. Но есть условие для отношения B (B2>50).

Поэтому первая команда для IO – A':

SELECT A1, A2 FROM A.

Аналогично предыдущему, выделим атрибуты отношения C.

В рассматриваемом примере – это C1, C2, C3. Условия селекции отсутствуют.

Соответственно вторая команда – C`:

SELECT C1, C2, C3 FROM C.

Таким образом, сформированы два промежуточных отношения – A` и C`, которые соединяются условием A1=C2.

Сформируем первую команду для JOIN – J`.

Общий вид команд соединения:

SELECT *атрибуты_отношения* FROM *имя_отношения1, имя_отношения2* WHERE *условие_соединения.*

Условие_соединения известно (A1=C2). *Имя_отношения1* – A`, *имя_отношения2* – C'.

Определим список атрибутов, используемых на более поздних стадиях обработки запроса. Оставшиеся еще не рассмотренными условия позволяют заключить, что из всех атрибутов отношений A` и C` в дальнейшей обработке принимают участие A2 (входит в конечный результат), C1 (участвует в операции соединения), C3 (входит в конечный результат).

Итак, команда J`:

SELECT A2, C1, C3 FROM A`, C` WHERE A1=C2

Следующей будет команда B`. Поступаем тем же способом, что и для A` и C`. Но не забудем, что в исходном запросе имеется условие селекции B2>50, которое нужно включить в команду.

Таким образом, команда B`:

SELECT B1 FROM B WHERE B2>50.

Атрибут B2 используется только в условии селекции и в дальнейшем не используется.

Очередная команда – это соединение J`` отношения B` и результата предыдущего соединения по условию B1=C1. Определим список атрибутов отношения J``. На заключительном этапе обработки потребуются только два атрибута A2 и C3. Оба входят в результат.

Команда J``:

SELECT A2, C3 FROM J',B' WHERE B1=C1.

В исходном запросе имеется операция агрегации SUM, которая выполняется на «логическом» процессоре SORT.

Общий вид команд для SORT:

SELECT атрибуты FROM имя_отношения
GROUP BY атрибуты_агрегации
ORDER BY атрибуты_сортировки.

Эта команда в большинстве случаев повторяет исходный запрос.

Команда S`:

SELECT A2, SUM(C3) FROM J`` GROUP BY A2.

В конечном итоге имеем:

Команды плана обработки запроса для узлов IO

A': *SELECT A1%2, A1, A2 FROM A*

C': *SELECT C2%2, C1, C2, C3 FROM C*

B': SELECT B1%2, B1 FROM B WHERE B2>50

Команды плана запроса для узлов JOIN

J' : *SELECT C1%2, A2, C1, C3 FROM A', C' WHERE A1=C2*

J'': SELECT A2%2, A2, C3 FROM J' , B' WHERE B1=C1

Команды плана запроса для SORT:

S': SELECT A2, SUM(C3) FROM J'' GROUP BY A2

Параллельная обработка запроса

Сформированные команды выполняются параллельно на соответствующих узлах кластера. Как уже отмечалось выше, отношения базы данных горизонтально распределены по узлам IO. При этом каждый узел независимо обрабатывает собственную часть данных.

Параллельный алгоритм соединения. Для выполнения операции соединения в системе применен параллельный алгоритм с использованием хеширования [12]. Хеширование в данной системе используется в двух разных случаях: 1) для распределения исходного отношения по множеству узлов хранения IO; 2) при реализации операции соединения.

Применение хеширования перед непосредственным *соединением* позволяет значительно сократить время выполнения этой операции. В процессе хеширования оба соединяемых отношения разделяются на непересекающиеся блоки отношений меньшего размера, над которыми может независимо (параллельно) выполняться операция соединения. При этом для каждой пары таких блоков выделяется отдельный процессор JOIN.

В данном случае функция хеширования применяется к атрибутам отношений, по которым выполняется соединение. Данный алгоритм рассчитан только на один тип операции соединения – соединения *по условию равенства атрибутов*.

Выполнение сформированного плана обработки по тактам. На следующих рисунках представлена реализация параллельного алгоритма обработки запроса для рассматриваемого примера.

Такт №1 *(выполнение команды A' на IO* – рис. 14*).*

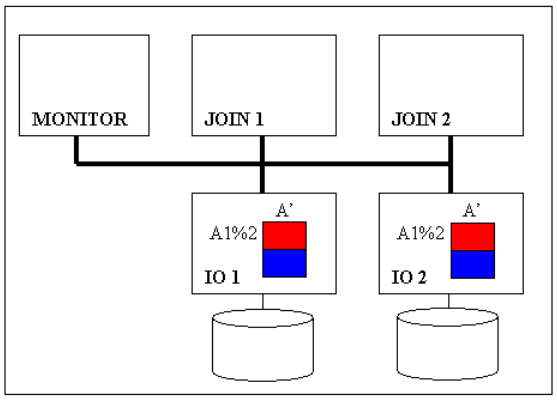

Рис. 14

Выполнение запроса начинается с команды: *A':*

SELECT A1%2, A1, A2 FROM A.

Выражение A1%2 реализует операцию хеширования, о которой было сказано выше. В качестве функции хеширования использован остаток от деления по модулю 2. Значение 2 в функции хеширования (2 узла JOIN) показывает, на сколько блоков разбивается результат выполнения команды A' перед операцией соединения. Если бы текущая конфигурация кластера содержала 4 узла JOIN, то функция хеширования выглядела как A1%4.

В результате выполнения команды A' каждая запись будет иметь признак, принимающий значение 0 либо 1. Этот признак указывает номер узла JOIN, на который должна быть переправлена данная запись.

Такт №2 *(динамическая сегментация на IO – по атрибуту A1)*. Перед отправкой результата выполнения команды A' на отдельные узлы JOIN выполняется сборка записей с одинаковым значением признака на соответствующие узлы IO (рис. 15).

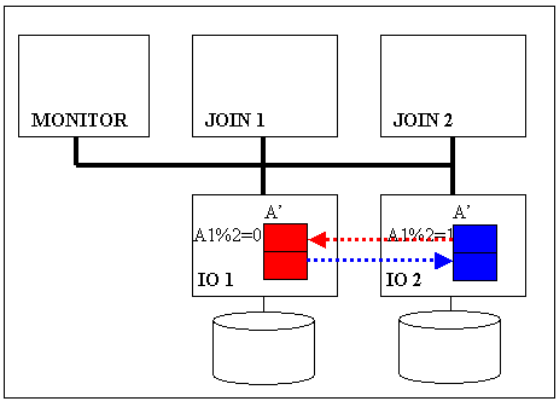

Рис. 15

Такт №3 *(пересылка первого промежуточного отношения с IO на JOIN)*. Полученные ранее блоки первого промежуточного отношения пересылаются узлам JOIN (рис. 16).

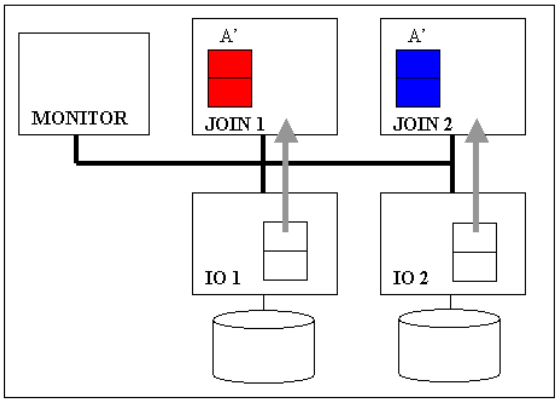

Рис. 16

Такт №4 *(выполнение второй команды на IO)*. После передачи результатов команды A’ узлы IO выполняют следующую команду плана обработки (команду B’) (рис. 17). Узлы JOIN ожидают результаты этого действия, чтобы начать выполнение соединения.

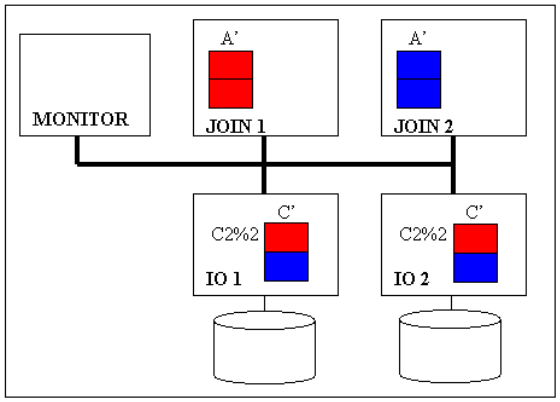

Рис. 17

Такт №5 *(динамическая сегментация на IO – по атрибуту C2).* Если результаты команды A’ хешировались по атрибуту A1, то результат команды C’ хешируется по атрибуту C2 (рис. 18). Именно по этим атрибутам (A1 и C2) будет выполняться операция соединения.

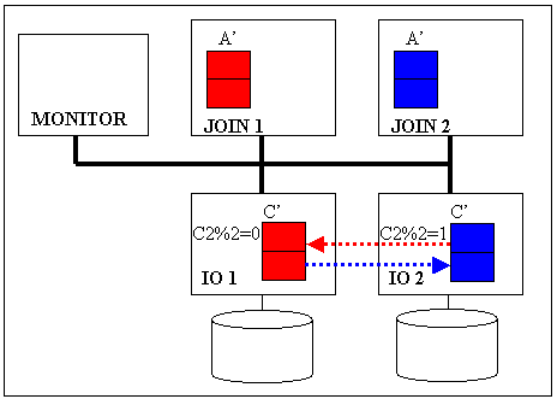

Рис. 18

Такт №6 *(пересылка второго промежуточного отношения на JOIN)*. На узлы JOIN пересылаются результаты выполнения команды C’ (рис. 19).

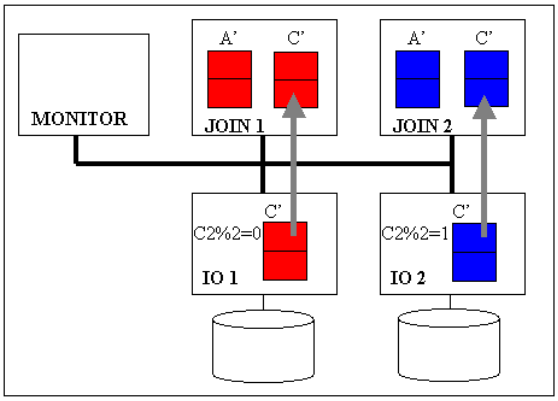

Рис. 19

Такт №7 *(Параллельное выполнение команд на IO и JOIN – внутризапросный параллелизм).* На данном такте параллельно выполняется очередной запрос на узлах IO и операция соединения на узлах JOIN (рис. 20).

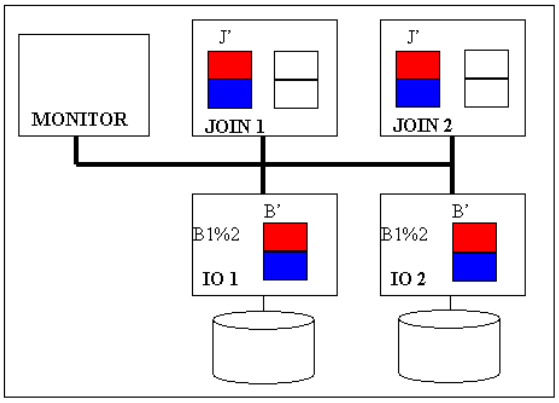

Рис. 20

Такт №8 *(динамическая сегментация: на IO – по атрибуту B1; на JOIN – по атрибуту C1).* Как видно из рис. 21, операция хеширования выполняется не только на IO, но и на JOIN. И там, и там происходит сборка записей, имеющих одинаковое значение функции хеширования .

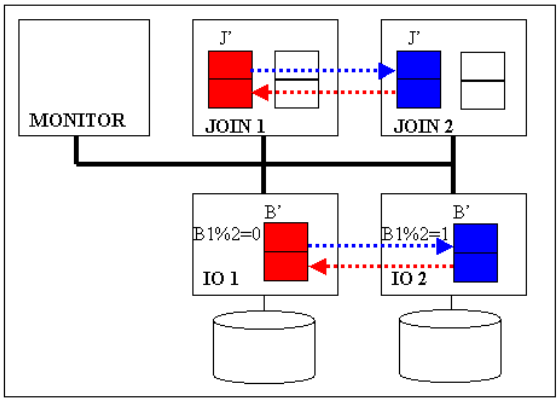

Рис. 21

Такт №9 *(пересылка третьего промежуточного отношения с IO на JOIN).* В этом такте на JOIN пересылается результат выполнения команды B’ (рис. 22).

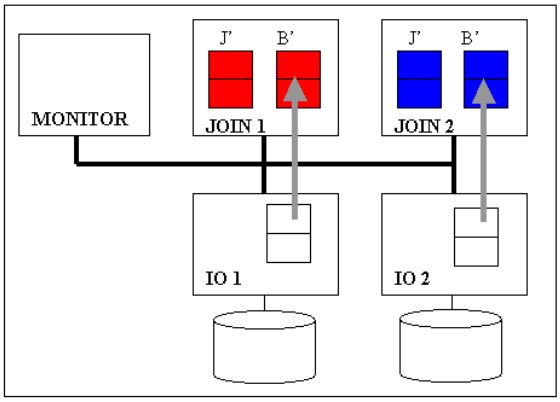

Рис. 22

Такт №10 *(выполнение второй команды на JOIN)*. Узлы JOIN приступают к выполнению команды J'', а узлы IO могут приступать к выполнению первой команды следующего запроса пользователя (помечен менее интенсивным цветом) (рис. 23).

Такт №11 *(выполнение команды на SORT)*. После выполнения последней операции соединения его результат пересылается в модуль SORT (находится на узле MONITOR) (рис. 24). Этот модуль сначала выполняет конкатенацию результатов с узлов JOIN, а затем –агрегацию и сортировку результата.

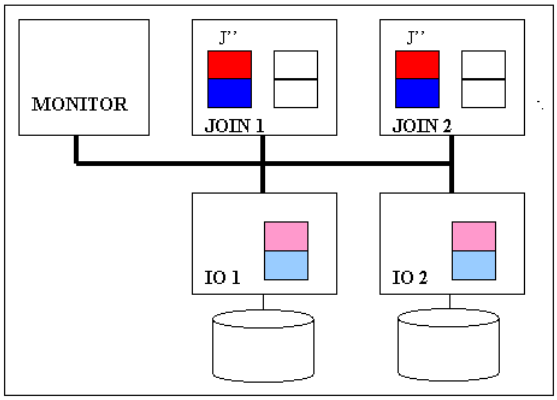

Рис. 23

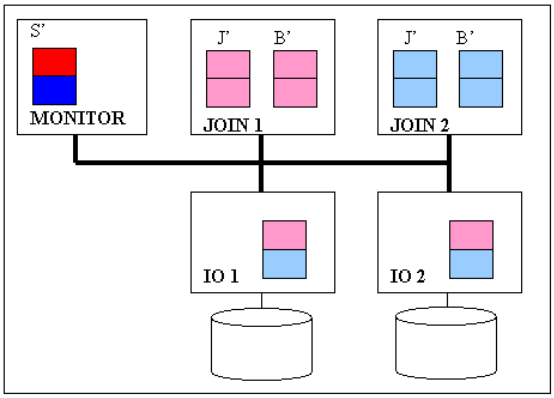

Рис. 24

На этом обработка запроса заканчивается, результат обработки передается пользователю.

Вопросы для самоконтроля*:*

1. Какие функции выполняет модуль irun СУБД Clusterix?
2. Какие функции выполняет модуль jrun СУБД Clusterix?
3. Какие функции выполняет модуль mlisten СУБД Clusterix?
4. Какой из способов распределения данных по узлам хранения кластера используется в СУБД Clusterix?
5. Какой тип архитектуры СУБД Clusterix изображен на рисунке (см. ниже)?
6. Какое обозначение имеет конфигурация, изображенная на рисунке (см. ниже)?
7. Что означает горизонтальное распределение отношения базы данных.
8. Какая операция реляционной алгебры выражается формулой: $\sigma_{\theta}(R)$?
9. Какая операция реляционной алгебры выражается формулой: $\pi_{A}(\sigma_{\theta}(R \times S))$?
10. Какие типы архитектур параллельных СУБД выделил Стоунбрейкер?

ЛИТЕРАТУРА

1. Озкарахан Э. Машины баз данных и управление базами данных: Пер. с англ. – М.: Мир, 1989.
2. *Ульман Дж. Основы систем баз данных – М.: Финансы и статистика, 1983.*
3. Калиниченко Л.А., Рывкин В.М. Машины баз данных и знаний. – М.: Наука, 1990.
4. Stonebraker M. The case for shared nothing // Database Engineering Bulletin. March 1986. Vol. 9. No 1. P.4-9.
5. Соколинский Л.Б. Обзор архитектур параллельных систем баз данных //Программирование. 2004. №6. С.1-15
6. Оззу М.Т., Валдуриз П. Распределенные и параллельные системы баз данных //СУБД. 1996. №4.
7. DeWitt D.J., Gray J. Parallel Database Systems: The future of high – performance database systems //Communications of the ACM. 1992 V.35. № 6. P.85-98.
8. http://www.mysql.com
9. Абрамов Е.В. Параллельная СУБД Clusterix. Разработка прототипа и его натурное исследование //Вестник КГТУ им. А.Н. Туполева. 2006. №2. С.52-55.
10. *Райхлин В.А. Моделирование машин баз данных распределенной архитектуры // Программирование. 1996. №2. С.7-16.*
11. Абрамов Е.В., Куревин В.В. Разработка и реализация претрансляции запросов для PC-кластеров баз данных //XIV Туполевские чтения. Материалы конференции. Том IV. Казань, 2006. С.39-40.
12. Kuitsuregawa M., Ogawa Y. A New Parallel Has Join Method with Robustness for Data Skew in Super Database Computer (SDC), Proceedings of the Sixteenth International Conference on Very Large Data Bases, Melbourne, Australia, August, 1990. P.210-221