Хранение и обработка сверхбольших объемов иерархических данных♣

© *Шабанов В.И.*

Рамблер Интернет Холдинг
vs@rambler-co.ru

Аннотация

В статье приводится описание метода, применяемого нами для хранения и распределенной обработки сверхбольших таблиц данных (сотни миллиардов записей). Наш метод оптимизирован для задач массовой параллельной обработки данных, а система хранения учитывает свойство внутренней иерархичности данных, встречаемое в большом классе задач. Мы описываем базовый набор примитивов нашей системы, разработанный нами формат данных HCS (Hierarchically Compressed Stream) и одноименную библиотеку для работы с этим форматом.

1 Предпосылки

Задачи поиска и анализа информации в больших неструктурированных массивах (поиск, тематическая классификация, кластеризация, извлечение фактов и т. д.) требуют обработки огромных объемов данных. Эти объемы нужно где-то хранить, както организовывать операции удаления, добавления и замены данных. Самое простое, но не самое эффективное решение, – поместить всю информацию в реляционную базу, такую, например, как Oracle. Более трудоемкое решение – разработать специализированный формат и написать соответствующую библиотеку.

Если сравнить типичные паттерны использования данных в системах поиска и обработки текстов с «идеальными» примерами использования SQL, то мы увидим следующую картину:

Система, для которой использование SQL - оптимум.	*Типичная система Information Retrieval & Web mining*
Имеет большое количество таблиц со сложными связями, индексами, триггерами и т. д.	Небольшое количество таблиц, но при этом некоторые таблицы содержат десятки или даже сотни млрд. записей.
В ходе типичной операции меняется относительно небольшое количество записей.	Характерны большие изменения. Например, при обновлении 10% поисковой базы PageRank меняется почти у всех страниц.
Активно используются сложные многоуровневые транзакции, многопользовательский доступ с параллельными изменениями, блокировками и т. д.	Программы монопольно обрабатывают все таблицы целиком, при этом использование транзакций для большинства таблиц бессмысленно. Максимум – полный откат базы на состояние, которое было перед запуском очередного этапа программы.
Накладные расходы на извлечение информации из базы (копирование памяти, обмен данными между процессами и т. д.) приемлемы, так как из СУБД за одну операцию извлекаются относительно небольшие наборы данных.	Мы читаем и/или записываем в базу настолько большое количество данных, что мириться с накладными расходами на транспортировку, преобразование форматов и т. д. невозможно.
Данные на дисках занимают разумный объем, оптимизировать затруднительно, т.к. никто не будет тратить время на специальный формат для каждой из сотен таблиц БД.	Нам совсем не безразлично, занимает индекс поисковой машины 5 Тб или 50 Тб. Мы готовы тратить время на специальные форматы, лишь бы сэкономить место и время на чтение/запись данных.

Итак, для задач поиска и обработки текстов необходимы специализированные базы данных, оптимизированные под реальные паттерны доступа информации.

2 Что нам надо

- система должна быть распределенной, допускать параллельную обработку данных на всех доступных компьютерах сети;
- мы не можем себе позволить «лишние» операции передачи данных от одной программы к

Труды 8ой Всероссийской научной конференции "Электронные библиотеки: перспективные методы и технологии, электронные коллекции" - RCDL'2006, Суздаль, Россия, 2006.

другой, поэтому функции доступа к данным должны встраиваться непосредственно в каждую из программ, выполняющих анализ информации;
- набор операций с данными и базовые примитивы должны быть оптимизированы под наши задачи;
- мы, тем не менее, не готовы отказаться от универсальности: периодически нам приходится добавлять или удалять таблицы, отдельные поля, менять их форматы и т. д.
- система должна быть надежной, содержать механизмы для самоверификации и восстановления в случае сбоев.

3 Чего нам не надо

- мы охотно откажемся от всех операций, которые увеличивают накладные расходы, замедляют или усложняют систему;
- придется также смириться с тем, что у нас не будет красивых и удобных визуальных инструментов для управления данными, подобных утилитам, входящим в комплект поставки любого SQL-сервера.

4 Известные решения

Проблемой параллельной обработки больших объемов данных занимаются с момента изобретения ЭВМ.

Существует очень много специализированных программных модулей для построения поисковых индексов. Самой часто цитируемой является книга Managing Gigabytes [1]. Еще один пример описан в [2]. Это хорошие, работающие системы, но они не обладают универсальностью, на них трудно построить что-нибудь кроме инвертированного индекса.

Другая категория – встроенные базы данных и библиотеки для хранения результатов научных экспериментов:

- *Berkeley DB* [3]. Исключительно удачная библиотека для сотен миллионов записей, но не для миллиардов. В Рамблере эта библиотека интенсивно используется, периодически обнаруживаются ситуации, когда структура БД Berkeley DB необратимо разрушается при сбое сервера, на котором она обрабатывается. Несмотря на это, индексы HCS-таблиц реализованы именно на этой БД. Мы можем себе это позволить, т.к. индексы всегда можно перестроить заново при наличии исходного HCS-файла.
- *HDF* от NSCA [4]. У этой библиотеки сложный интерфейс доступа к данным и проблемы с организацией быстрого последовательного доступа к данным.
- *Metakit* [5]. Интересное решение, но, к сожалению, ограниченный фиксированный набор типов данных и за счет сложной структуры данных (и возможности rollback) сильно снижена производительность.

Следующая категория: библиотеки параллельного программирования. Наиболее распространенные основаны на интерфейсе *MPI* (*LAM* [6], *MPICH* [7] и т. д.). Базовые примитивы *MPI* хороши для распараллеливания математических расчетов, однако для того, чтобы провести быструю параллельную обработку очень большого количества записей, необходимо разработать поверх интерфейса *MPI* специализированный слой, занимающийся асинхронной транспортировкой данных. По сложности он оказался сравним с самим кодом демона *MPI*, и, плюс к этому, мы столкнулись с неустойчивостью и низкой производительностью реализаций *MPI* в условиях реальной эксплуатации. Поэтому от использования *MPI* было решено отказаться.

И, наконец, последняя категория – большие, сложные системы массовой обработки, такие как *MapReduce* [8], *Condor* [9]. Все они исключительно сложны в реализации и при этом отнюдь не дешевы с точки зрения потребляемых ресурсов. Во время проектирования одной из компонент поисковой системы мы пробовали построить ее на примитиве *MapReduce* и оказалось, что как с точки зрения времени разработки, так и по размеру дисковой памяти/сетевого трафика это решение отнюдь не является оптимальным. К тому же *MapReduce* решает только проблему обработки данных и ничем не помогает с организацией эффективного структурированного механизма хранения.

5. HCS: Hierarchically compressed stream

HCS – это набор иерархически сжатых реляционных таблиц, хранимых на группе серверов, объединенных одной локальной сетью (кластере).

Система обработки информации, использующая HCS, построена на следующих принципах:

- Основная операция – последовательное чтение одной или нескольких таблиц с выдачей результата в виде новых HCS-таблиц.
- Все HCS-таблицы отсортированы одним единственным способом: лексикографически по возрастанию (сначала по самой левой колонке, затем по следующей и т. д.). Некоторые и них могут быть снабжены индексным файлом для быстрого поиска записей по заданному префиксу.
- Если нам нужен какой-то другой способ сортировки, то мы помещаем в поля HCS-таблицы специальным образом измененные значения. Например, для сортировки по убыванию времени можно положить в первую колонку «антивремя»: (*0xFFFFFFFF – time*)♣.
- Отсортированные данные обычно легко сжимаются, т.к. значения в левых колонках таблицы меняются реже, чем в правых. HCS автоматически учитывает этот факт и сжимает информацию.
- Для того, чтобы распараллелить обработку, программа запускается на всех машинах кластера,

получая на вход фрагменты HCS-таблицы. После этого выполняется объединение выходных HCS-таблиц и сохранение результата в кластере.

- В случае аварии на одной из машин автоматически выполняется перезапуск программы на соответствующие порции данных. При этом возможна автоматическая миграция процесса на другой сервер кластера.
- Система реализует следующие примитивы:
  - **CAT**: последовательное чтение одного или нескольких входных HCS-потоков.
  - **GREP/FILTER**: чтение из входного HCS-потока записей, начинающихся с заданного префикса (заданных значений первых *n* полей таблицы). Данная операция выполняется быстрее, чем последовательное чтение, так как HCS знает о том, как сжаты данные, и поэтому умеет быстро пропускать куски таблицы с «неинтересными» значениями первых *n* полей. При наличии предварительно построенного индекса операция идет еще быстрее.
  - **MERGE**: слияние нескольких отсортированных HCS-потоков в один.
  - **JOIN**: параллельное чтение нескольких входных потоков с объединением или пересечением списков. Семантика операции аналогична INNER/OUTER JOIN в SQL.
  - **UNIQUE**: удаление одинаковых строк из входного или выходного потока.
  - **SORT**: сортировка выходного HCS-потока.
  - **SPLIT**: разрезание выходного потока на части по заданному критерию.
  - **PUSH**: пересылка частей выходного потока на соответствующие сервера кластера.

Как можно увидеть из списка операций, HCS не поддерживает операций вставки/удаления записей из таблицы и замены значений в записях. HCS допускает только массовые операции. В некоторых приложениях эти операции все же имеются, но реализованы исключительно на основе примитивов **SORT**, **MERGE** и **JOIN.**

Большая часть приложений рассматривает HCS-таблицу именно как поток данных, последовательно читая записи и выполняя их обработку. Поэтому в таких приложениях совсем не обязательно хранить ее на диске: мы можем организовать конвейер аналогичный потокам ввода-вывода unix. Например, если мы хотим генерируемые программой данные разрезать на несколько частей, каждую из них отсортировать и отправить на соответствующие машины вычислительного кластера, достаточно «сцепить» потоки данных HCS: **SPLIT** | **SORT** | **PUSH.**

Сама библиотека HCS представляет собой набор шаблонов C++, с помощью которых программа может удобно читать/писать данные и набор утилит, обеспечивающих параллельный запуск, верификацию и аварийное восстановление запущенной программы.

6. Примеры использования:

6.1 Простой пример.

Пусть мы имеем N серверов, на которых хранятся логи (протоколы работы) поисковой машины. Найдем при помощи HCS все часто встречающиеся пары запросов, которые пользователи подают в течение одной сессии. Для этого заведем 3 таблицы:
Таблица *UserQueries* – первая промежуточная таблица

User_id	Идентификатор пользователя
Unixtime	Дата поискового запроса
Query	Текст поискового запроса

Таблица *QueryPairs* – вторая промежуточная таблица

Query1	Текст первого запроса
Query2	Текст второго запроса
User_id	Идентификатор пользователя, в поисковой сессии которого встретилась эта пара запросов

Таблица *QueryPairsResult* – окончательный результат

Query1	Текст первого запроса
Query2	Текст второго запроса

Первую таблицу разрежем на M_1 частей и поместим на кластер из M_1 серверов, вторую – на M_2 частей и поместим на соответствующее количество серверов кластера M_2. На самом деле, никаких ограничений по разрезанию и хранению нет, мы можем использовать для хранения обеих таблиц одни и те же сервера. В обоих случаях выберем следующий критерий разрезания: номер сервера для хранения и обработки записи равен

CRC32(первое_поле_HCS-таблицы) % M_i

Для решения задачи нам нужно после получения очередной порции логов выполнить следующие операции:

1. Параллельно на всех *N* машинах выполняем следующие операции:
   a. Читаем лог, пишем локальную таблицу *UserQueries*. Во время записи выполняем операции **SPLIT** (разрезаем таблицу на M_1 частей).
   b. Выполняем для каждой сгенерированной части операцию **SORT**. На самом деле, библиотека позволяет обе операции выполнять одновременно, чтобы минимизировать объем сохраняемых на диске промежуточных данных.

2. Запускаем операцию **PUSH**: доставляем сгенерированные порции данных со всех *N* серверов на машины кластера M_1. Всего мы транспортируем $N^* M_1$ порций.
3. Параллельно на всех машинах кластера M_1 делаем следующее:
   a. Выполняем операцию **MERGE**: сливаем «большие» HCS-таблицы *UserQueries* с доставленными на этапе 2 порциями (на каждой из M_1 машин у нас получается *N* кусочков).
   b. Одновременно с записью выходного HCS-файла формируем дополнительно кусочек таблицы *QueryPairs*, содержащий все пары запросов, поданные хотя бы одним пользователем в течение короткого интервала времени. Данных для этого у нас вполне достаточно: все запросы одного пользователя идут подряд, так как HCS-таблица *UserQueries* отсортирована, для каждого пользователя список запросов хронологически упорядочен, потому, что во второй колонке хранится время. Формируемые данные проходят через **SPLIT** (мы их разрезаем на M_2 частей), **UNIQ** и **SORT**.
4. Запускаем операцию **PUSH**: доставляем сгенерированные порции данных со всех серверов кластера M_1 на соответствующие сервера кластера M_2. Всего у нас получается $M_1 * M_2$ порций.
5. Параллельно на всех машинах кластера M_2 делаем следующее:
   a. Выполняем операцию **MERGE**: сливаем доставленные на предыдущем этапе M_1 порций таблицы *QueryPairs*. Полную таблицу *QueryPairs* никуда не пишем, просто анализируем и выделяем все пары, которые подало достаточно большое число пользователей. Снова у нас достаточно информации для принятия решения: пары сначала отсортированы по первому запросу, затем по второму, а уж затем – по идентификатору пользователя. Поэтому все пользователи, подавшие одну пару запросов, находятся рядом и нам достаточно просто посчитать их количество.
   b. Пишем выходной HCS-файл *QueryPairsCount* со списком частых пар запросов
6. Сливаем M_2 файлов *QueryPairsCount* в один большой файл.

В реальном приложении этапы 1.b, 2, 3.a, 4, 5.b, 6 представляют собой вызовы библиотечных функций или запуск скриптов, а остальные очень компактны и просты, т.к. вся их работа сводится к итерации по входному потоку данных, выполнению простой трансформации и записи в выходной поток.

6.2 Более сложный пример

Данный пример можно усложнить и оптимизировать. Например, можно все тексты запросов вынести в отдельную таблицу *QueryTexts*, а соответствующие поля в описанных выше таблицах *UserQueries* и *QueryPairs* заменить значениями хэш-функций. В этом случае мы сможем существенно уменьшить количество обрабатываемых и передаваемых по сети данных.

Итак, у нас теперь 6 таблиц:

Таблица *QueryHashes* – словарь текстов поисковых запросов. Из порядка полей очевидно, что она упорядочена по значениям хэшей.

QueryHash	Хэш-функция от текста поискового запроса
QueryText	Текст поискового запроса

Таблица *UserQueries* – первая промежуточная таблица

User_id	Идентификатор пользователя
Unixtime	Дата поискового запроса
QueryHash	Хэш-функция от текста поискового запроса

Таблица *QueryPairs* – вторая промежуточная таблица

QueryHash11	Хэш первого запроса
QueryHash22	Хэш второго запроса
User_id	Идентификатор пользователя, в поисковой сессии которого была эта пара запросов

Таблица *QueryPairsResult0* – результат в хэшах

QueryHash1	Хэш первого запроса
QueryHash12	Хэш второго запроса

Таблица *QueryPairsResult1* – Частично оттранслированный результат

QueryHash1	Хэш второго запроса
QueryHash2	Текст первого запроса

Таблица *QueryPairsResult* – окончательный результат

Query1	Текст первого запроса
Query2	Текст второго запроса

Таблица *QueryHashes* у нас будет хранится на кластере из M_3 серверов. Критерий разрезания – остаток деления *QuryHash* на M_3.

Оптимизированный алгоритм выглядит так:

1. Параллельно на всех *N* машинах выполняем следующие операции:
   a. Вход – лог, выходные HCS-потоки: **SPLIT** | **SORT** (*UserQueries*) и **SPLIT** | **SORT** (*QueryHashes*).

2. Запускаем операцию **PUSH**: доставляем разрезанные отсортированные порции *UserQueries* на кластер M_1, а разрезанные отсортированные порции *QueryHashes* – на кластер M_3.
3. Параллельно на всех машинах кластера M_1 делаем следующее:
   a. Вход: **MERGE** (*UserQueries*). Выход: обновленная таблица *UserQueries* и **UNIQ** | **SPLIT** | **SORT** (*QueryPairs*) .
4. Одновременно с шагом 3 параллельно на всех машинах кластера M_3 делаем следующее:
   a. Вход: **MERGE** (QueryHashes). Выход: обновленная таблица QueryHashes.
5. Запускаем операцию **PUSH**: доставляем разрезанные отсортированные порции *QueryPairs* на соответствующие сервера кластера M_2.
6. Параллельно на всех машинах кластера M_2 делаем следующее:
   a. Вход: **MERGE**(*QueryPairs*), выход: **SPLIT**(*QueryPairsResult0).* Результат разрезается по хэшам от текстов запросов, чтобы мы могли оттранслировать их обратно в тексты.
7. Запускаем операцию **PUSH**: доставляем *QueryPairsResult0* на сервера кластера M_3.
8. Параллельно на всех машинах кластера M_3 делаем следующее:
   a. Вход: **JOIN** (**MERGE**(*QueryPairsResult0*), *QueryHashes*)
   Выход: **SPLIT | SORT**(*QueryPairsResult1).* Здесь мы параллельно идем по двум одинаково отсортированным таблицам и заменяем один из хэшей строками. Для того, чтобы эту операцию можно было повторить для второго хэша, меняем поля местами, затем снова разрезаем и сортируем.
9. Запускаем операцию **PUSH**: переносим разрезанные отсортированные порции *QueryPairsResult1* между всеми серверами кластера M_3.
10. Параллельно на всех машинах кластера M_3 делаем следующее:
    a. Вход: **JOIN** (**MERGE**(*QueryPairsResult1*), *QueryHashes*)
    Выход: (*QueryPairsResult).* Здесь мы получаем результат уже в строках.
11. Сливаем M_3 файлов *QueryPairsResult* в один большой файл.

Как видно из примера, реализованный в HCS набор примитивов позволяет выполнить сложные параллельные трансформации любых объемов данных. В реальном приложении – поисковой машине Рамблер в виде HCS-таблиц представлены прямой и обратный графы гиперссылок, ссылочный индекс и другие части. За ночь система успевает параллельно на большом кластере выполнить длинную цепочку eval1 | **SPLIT** | **SORT** | **PUSH** | **MERGE** | **JOIN** | *eval2* | **SPLIT** | **SORT** | **PUSH** | **MERGE** | eval4 и т. д., где evalN – алгоритм, выполняющий относительно простую трансформацию данных.

7. Остальные свойства HCS

В ходе разработки и внедрения нам пришлось столкнуться с серьезными проблемами, связанными с реальной эксплуатацией системы на больших кластерах из относительно дешевых серверов. Другая часть проблем связана с инертностью: время формирования данных достаточно велико, и поэтому при внесении изменений в код (например, добавлении дополнительных полей) нам приходилось сталкиваться с тем, что еще несколько недель после добавления программы получали на вход «старые» данные без новых полей. В результате HCS получил следующие свойства:

1. Информация о структуре файла и способах хранения данных храниться в самом файле (принцип управления данными), а не в отдельных конфигурационных файлах, и, тем более, не в исходных текстах программ. Это снижает риск десинхронизации файлов данных и файлов конфигурации.
2. Данные хранят дополнительную информацию для самоверификации (на случай порчи диска или ошибок при передаче). Библиотека самостоятельно подсчитывает контрольные суммы читаемых данных и проверяет порядок сортировки записей в таблицах.

8. Внутреннее устройство HCS

Предположим, у нас имеется таблица с H строками (в нашем примере H=8):
Пара (X,Y) идентифицирует ячейку данных на уровне X=0÷(H - 1). Y нумерует ячейки в пределах одного уровня. Пусть на одном уровне находятся данные одного типа.

(0,0)	(1.0)	(2,0)	(3,0)
(0,0)	**(1.0)**	**(2,0)**	(3,1)
(0,0)	**(1.0)**	(2,1)	(3,2)
(0,0)	**(1.0)**	**(2,1)**	(3,3)
(0,0)	**(1.0)**	(2,1)	(3,4)
(0,0)	(1.1)	(2,2)	(3,5)
(0,0)	**(1.1)**	**(2,2)**	(3,6)
(0,0)	**(1.1)**	(2,3)	(3,7)

Жирным шрифтом выделены ячейки, которые повторяются, т.е. занимают лишнюю память при хранении «в лоб».

Определим коэффициент избыточности как:

$$K_{izb}=\text{frac}\,(N, M)\ ,$$

где N – количество ”лишних” ячеек таблицы, а M – размер таблицы. В нашем примере K_{izb}=frac (17, 4 ×8)≈0.53, т.е. больше половины. Применительно к реальным данным коэффициент избыточности может достигать 0.6÷0.8.

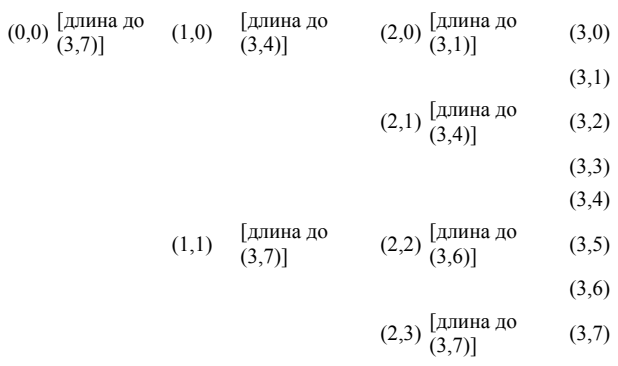
Существует другой способ: организовать таблицу в виде цепочки из H реляционных таблиц, в которой каждая следующая таблица связана внешним ключом с предыдущей. В этом случае тоже возникает проблема избыточности, так как приходится хранить значения самих ключей, размер которых

зачастую сопоставим с размерами самих ячеек. Кроме того, при этом существенно страдает производительность.

Библиотека HCS автоматически находит повторяющиеся строки и кодирует их так, чтобы они занимали минимальный объем. Последовательности строк таблицы, в которых первые *n* колонок не меняют своих значений, называются секциями n-ого уровня. Таким образом, вся таблица представляет собой последовательность секций 0-ого уровня, каждая из которых состоит из значения первой колонки и последовательности секций 1-ого уровня, которые устроены точно так же.

8.1 Физический формат иерархии

HCS – это совокупность вложенных секций и подсекций. Для того, чтобы уложить эту систему в последовательность байт, в начале каждой секции пишется ее размер. Это позволяет при необходимости быстро перемещаться вперед к следующей секции, анализируя только первую строку секции, что существенно сокращает временные затраты на построение индекса или слияния с фильтрацией. Для описанного выше примера таблицы HCS-файл выгдядит следующим образом:



8.2 Схема HCS

Каждая HCS-таблица обладает *схемой,* которая описывает структуру записей таблицы. Запись состоит из набора полей, каждое из которых обладает названием, типом, размером (в байтах), а также признаком того, является ли оно полем переменной длины или нет. Вот пример схемы ссылочного графа:

```
{src}C*1024:{date}T4:{trgt}C*1024:{ref}C*256
```

Данная схема состоит из четырех полей. В фигурных скобках – названия полей. Буква после названия – тип поля; символ * означает, что поле имеет переменную длину; число после типа – длина поля (или максимальная длина поля, если поле – переменной длины).

8.3 Загрузка данных в программу C++

При чтении таблицы, строки прямо из HCS-таблицы попадают в C-структуру без лишних преобразований. При записи, строки попадают прямо из C-структуры в HCS-таблицу.

Такое поведение достигается за счет статического связывания полей C-структуры с полями HCS-таблицы (пример приводится ниже).

Это позволяет обращаться к полям по имени полей структуры, что значительно улучшает читабельность программ.

8.4 Основные объекты библиотеки

Основными объектами библиотеки являются классы-шаблоны HCS::Input, HCS::Output и HCS_Output_auto. Первый применяется для чтения HCS-таблиц, а остальные два – для записи.

Метод int HCS::Input::get (struct_t *) читает следующую строчку из таблицы.

Метод void HCS::Output::put(int level, struct_t *) пишет строчку в файл начиная с уровня level.

Метод void HCS::Output_auto::put(struct_t *) пишет строчку, автоматически определяя границы секций.

Помимо этих двух классов были написаны другие *Input*- и *Output*- классы, имеющие подобный интерфейс и принимающие в качестве параметров шаблона другие *Input*- и *Output*- классы. Например, существуют *Output_sort*, *Output_uniq*, *Output_split*, *Output_filter*, *Input_merge* и другие специфические классы.

Подобная организация библиотеки позволила свести большинство задач к созданию на старте программы одного или нескольких *Input*- и *Output*-классов, а затем в тривиальном чтении данных из входных потоков и записи в выходной.

Более того, эта организация позволяет выйти за пределы HCS и написать посторонние *Input*- и *Ouput*- классы, которые обрабатывают данные другой природы. Например, были написаны классы *Output_dump* и *Input_undump*, переводящие HCS-файлы в текстовое представление и обратно.

9. Примеры кода HCS

Ниже приводятся примеры решения самых базовых задач, и, где это возможно, их аналог на SQL.

9.1 Декларирование новой таблицы

Вот пример того, как в HCS задается новая таблицы и ее связь с C-структурой.

```
SQL
CREATE TABLE refgr (
  src VARCHAR(1024),
  date DATETIME,
  dst VARCHAR(1024),
  ref VARCHAR(256)
);
```

```
HCS
#define STRUCTNAME   \
          refgr_struct
struct STRUCTNAME {
  char * src;
  hcs_len_t src_len;
  hcs_date_t date;
  char * trgt;
  hcs_len_t trgt_len;
  char * ref;
  hcs_len_t ref_len;
};
SOP_FIELDS_BEGIN(4)
  VARLEN_FIELD
    (src,src_len,1024),
  VARLEN_FIELD
```

```
    (trgt,trgt_len,1024),
  FIXLEN_FIELD(date),
  VARLEN_FIELD
    (ref,ref_len,256)
SOP_FIELDS_END

refgr_scheme_str =
   "{src}C*1024:{date}T4:
    {trgt}C*1024,
    {ref}C*256";
```

9.2 Запись в таблицу

В HCS-таблицу (в отличие от SQL) нельзя дописывать. Можно только создать новый файл и записать его целиком.

```
SQL
INSERT INTO refgr
  VALUES (
  "http://src.ru/",
  "http://trgt.ru/",
  some_date,
  "На главную"
);
```

```
HCS
HCS::Output<refgr_struct>
   output
       (refgr_scheme_str,
        "refgr.hcs" );
HCS::Struct_ptr
   <refgr_struct> p;
p->src="http://src.ru/";
p->trgt="http://trg.ru/";
p->date=hcs_date( );
p->ref="На главную";
output->put( p );
```

9.3 Чтение таблицы

```
SQL
SELECT * FROM
refgr
```

```
HCS
HCS::Input
    <refgr_struct>
      input ("refgr.hcs");
HCS::Struct_ptr
    <refgr_struct> p;

While (input.get (p)
       != HCS_EOF)
{
  // do something
};
```

9.4 Слияние сортированных HCS-файлов

Если исходные HCS-таблицы отсортированы, то для получения результата слияния достаточно создать еще один HCS-поток:

```
Input_merge<struct_t,Input<struct_t> >
    input( input_filenames );

Output<struct_t> output (input.scheme (),
                         output_filename );

pipe_manuallevel (input, output);
```

9.5 Сортировка слиянием

Сортировка без выбрасывания идентичных строчек.

```
Input_multi<struct_t, Input<struct_t> >
       input( input_filenames );
Output_sort<struct_t,
            Output<struct_t>,
            Output_auto<struct_t> >
      output (input.scheme( ),
              output_filenames);
pipe_autolevel (input, output);
```

Второй параметр шаблону *Output_sort* определяет *Output*, с помощью которого будут писаться выходные данные. Третий параметр – *Output*, с помощью которого пишутся промежуточные файлы, которые потом сливаются вместе.

Заключение

Нам удалось реализовать систему параллельного хранения и обработки данных, которая оптимизирована под задачи массового анализа сверхбольших объемов данных. Система обладает следующими свойствами:

- высокопроизводительная: мы удалили те функции «обычной» СУБД, без которых мы можем обойтись, используем бинарный формат представления данных и сжатие;
- полная: на HCS потенциально можно перенести почти все структуры данных поисковой машины и других сервисов Рамблера;
- удобная: мы тратим минимум времени на программирование рутинных операций и распараллеливание вычислений;
- масштабируемая: задачу можно «размазать» по большому количеству серверов с автоматическим восстановлением в случае сбоев.

К библиотеке был разработан обширный инструментарий для работы с БД. Наше решение позволило нам создать распределенную базу данных из нескольких сотен миллиардов записей и использовать ее для решения многочисленных задач.

Литература

1. Managing gigabytes: compressing and indexing documents and images, Witten, Moffat, and Bell, Van Nostrand Reinhold, 1994,
2. Building a Distributed Full-Text Index for the Web. Melnik, Sergey; Raghavan, Sriram; Yang, Beverly; Garcia-Molina, Hector, 2000. http://dbpubs.stanford.edu/pub/2000-29
3. Berkley. Berkley DB Database. http://www.sleepycat.com.
4. HCSA. Hierarchical Data Format. http://hdf.ncsa.uiuc.edu/.
5. Equi4. Metakit. http://www.equi4.com/metakit.html.
6. LAM/MPI Parallel Computing: http://www.lam-mpi.org/
7. MPICH2: http://www-unix.mcs.anl.gov/mpi/mpich2/
8. MapReduce: Simplified Data Processing on Large Clusters. Jeffrey Dean, Sanjay Ghemawat, 2004. http://labs.google.com/papers/mapreduce-osdi04.pdf
9. Condor Project: http://www.cs.wisc.edu/condor/

HCS: Organize and process huge amount of hierarchically organized data in parallel environment

Vladislav I. Shabanov

We describe an approach to organize, store and process large amount of structured data in cluster envi-

ronment. Specially designed set of primitives used to simplify and scale processing. Our library effectively compresses the data, splits it between cluster nodes and organizes fast information exchange between nodes. All data processing consists of compact chunks of code reading and writing data from/to specially designed streams. These streams can store information, distribute it into group of cluster nodes, perform sorting, filtration and other operations.

HCS library specially optimized for data usage patterns in Information Retrieval, Data Mining and other large scale Web data analysis tasks.

♣ Если unixtime –время в секундах, отсчитываемое от сотворения операционной системы Unix (1 января 1970 года), то (0xFFFFFFFF – unixtime) – количество секунд, которое осталось до наступления конца света в Unix.