Применение технологий OLAP и MapReduce для обработки результатов нагрузочного тестирования

Сенов А.А.

Костромской государственный технологический университет
г. Кострома, ул. Дзержинского, 17
andrey.senov@gmail.com

Аннотация. Современные трейдинговые платформы представляют собой сложные распределённые системы, обслуживающие большое число пользователей по различным протоколам передачи данных. При нагрузочном тестировании таких систем возникает существенный объём информации, которая может быть проанализирована QA-специалистами с целью обнаружения дефектов. Для этого часто требуется выявлять зависимости между значениями показателей, полученными в ходе тестирования, путём проведения многофакторного анализа, что при большом объёме данных является трудоёмкой задачей, требующей специализированного ПО, в частности, систем интерактивной аналитической обработки OLAP. В данной статье описывается оригинальный алгоритм многомерного анализа результатов нагрузочного тестирования на стороне клиента. С помощью технологии MapReduce предложенный алгоритм оптимизирован для работы под управлением многоядерных процессоров. *Ключевые слова:* OLAP, MapReduce, нагрузочное тестирование, анализ данных

1 Введение

Анализ результатов нагрузочного тестирования трейдинговых платформ в большинстве случаев связан с обработкой существенных объёмов информации. Это обусловлено тем, что современные платформы являются сложными распределёнными системами, обрабатывающими свыше 40 тысяч заявок в секунду при среднем времени отклика менее 300 микросекунд. С другой стороны, аппаратная инфраструктура тестовых инструментов по производительности на несколько порядков уступает биржевой. Это неизбежно влечёт необходимость разрабатывать инструменты как можно более простые, не перегруженные функциональностью, для достижения высокой эффективности при решении основных задач[1]. В связи с этим при анализе данных, полученных в ходе нагрузочного тестирования, чтобы быстро ответить на

поставленные вопросы, возникает потребность в использовании специализированного ПО, к какому можно отнести большой класс продуктов бизнес-аналитики BI (англ. *Business Intelligence*). Среди программных решений BI-класса можно выделить системы интерактивной аналитической обработки OLAP (англ. *Online Analytical Processing*), предназначенные для построения обобщённой информации на основе данных, имеющих многомерное представление. При нагрузочном тестировании такие системы помогают QA-аналитикам (англ. *Quality Assurance*) качественно осуществить многофакторный анализ зависимостей между значениями показателей, полученными в ходе тестирования, с целью обнаружения дефектов платформы.

2 Существующие Решения

В основе любой OLAP-системы лежит многомерная модель данных, называемая многомерным кубом, гиперкубом или просто кубом. По месту нахождения машины, рассчитывающей многомерные кубы, современные OLAP-системы можно разделить на две группы: клиентские и серверные. В случае клиентской системы сервер отправляет клиенту исходные данные, клиент производит расчет многомерных кубов и выдает результат пользователю. В случае серверной системы сервер рассчитывает многомерные кубы и отправляет результат клиенту, клиент выдает результат пользователю[2].

На сегодняшний день обе группы представлены на рынке широким спектром продуктов. Для серверных систем это, например, такие решения, как Microsoft SQL Server Analysis Services[3], OLAP Option to Oracle Database[4] или Palo[5]. Серверные решения являются наиболее быстрыми, надежными и имеют ряд других преимуществ: например, техническую поддержку от производителей. Как следствие, такие системы являются дорогостоящими. Можно принять во внимание наличие open-source версии Palo, но она имеет существенные ограничения по сравнению с коммерческой: например, отсутствие возможности использования более чем одного ядра процессора[6].

Среди клиентских решений популярными остаются электронные таблицы MS Office Excel, позволяющие выполнять многомерный анализ с помощью механизма Pivot table[7]. Pivot table удобно использовать, когда исходные данные изначально заносятся в электронную таблицу, что при серьёзных задачах, таких как нагрузочное тестирование, исключено. Поэтому пользователь обязательно сталкивается со сложностью конфигурации подключения к внешним источникам данных.

Факт наличия на рынке разнообразных продуктов, обладающих своими достоинствами и недостатками, говорит о том, что для OLAP, как и для многих сложных задач, до сих пор нет единого решения, одинаково подходящего для всех. В рамках статьи предлагается рассмотреть оригинальный алгоритм многомерного анализа результатов нагрузочного тестирования на стороне клиента. Решение главным образом направлено на сокращение затрат на тестовые и аналитические инструменты.

3 Предлагаемое Решение

В случае тестирования биржевых платформ существует несколько основных источников, из которых можно получить информацию для анализа:

1. Данные, которыми оперируют непосредственно тестовые инструменты;
2. Статистическая информация из внутренней базы данных платформы, а также записи журналов активности её компонентов и пользователей;
3. Независимые данные, полученные напрямую от сетевых устройств, через которые происходит общение клиентов с платформой.

Опыт показывает, что наиболее достоверным является источник под номером 3, но для полноты картины поведения тестируемой системы данные обычно собираются изо всех трёх источников и складываются в общую базу. Данные, как правило, представляют собой разнообразные сведения о типах сообщений, значениях полей сообщений, клиентах, разного рода настройках и, конечно, временных отметках. Опираясь на весь этот массив информации, аналитики оценивают такие параметры, как время отклика, количество сообщений, пропускную способность, используемую память, загрузку процессоров и прочее[8].

При создании алгоритма основным принципом служила простота, и вся его суть сводится к выполнению операций над generic-контейнерами в оперативной памяти клиентской машины. Перенос вычислений на сторону клиента обусловлен высокой производительностью современных настольных ПК, сочетающейся с их низкой стоимостью. Так, например, в настоящее время конфигурация среднего рабочего места это: процессор с 2-4 ядрами и 4-8 ГБ оперативной памяти. Такие характеристики в сочетании с гигабитными сетями, использующимися уже повсеместно, позволяют решить задачу многомерного анализа без привлечения дополнительных ресурсов.

Алгоритм написан на языке C++ с применением Qt Framework, что даёт такие преимущества, как высокое быстродействие, кроссплатформенность, а также возможность использования настольной реализации технологии MapReduce[9] с целью получения масштабируемого решения для работы под управлением многоядерных процессоров.

Как было сказано выше, в основе любой OLAP-системы лежит гиперкуб. И первая проблема, которую необходимо решить, – это представление гиперкуба с точки зрения кода. Для этого сначала необходимо определиться, как информация будет попадать из базы данных в гиперкуб. Ответ очевиден – нужно выполнить SQL-запрос SELECT. Запросы предлагается писать по правилам, которые отражены в листинге 1.

Листинг 1. Общий вид SQL-запроса выборки данных; где dimension0 … dimensionN – значения измерений гиперкуба, measure – мера, значение ячейки гиперкуба, образуемое при уникальном сочетании значений измерений.

```
select dimension0, dimension1, dimension2, …,
dimensionN, measure
from … inner join … on …
…
where <condition>
…
```

Проще говоря, запросы должны возвращать кортежи, состоящие из различных сочетаний измерений и соответствующие этим сочетаниям числовые значения анализируемой величины. Таким образом, правила помогают подготовить данные и передать их клиенту уже в структурированном виде.

Атрибуты кортежей, описывающие значения измерений, могут иметь самые разнообразные типы, а последний атрибут *measure* в общем случае, является числом с плавающей точкой. Так как на этом этапе задача состоит не только в создании универсального способа хранения данных в ОЗУ, но и обеспечении возможности построения простых и составных ключей по значениям измерений на осях гиперкуба, которые будут необходимы для упорядочивания и поиска информации, все измерения следует привести к одному типу ещё до размещения в ОЗУ. В качестве такого типа подходят строки. Это объясняется тем, что большинство измерений изначально имеют строковые значения: названия протоколов, сообщений, финансовых инструментов, имена пользователей и т.п., а все другие типы при необходимости всегда можно привести к строковому, используя SQL-функцию *cast*. Что касается дат, то при переводе их в строковый тип они приобретают ISO-формат *YYYY-MM-DD*, и упорядочивание таких строк напрямую соответствует упорядочиванию дат.

Таким образом, кортежи запроса можно описать следующим классом:

Листинг 2. Представление кортежей запроса, где поле класса _keys содержит атрибуты, представляющие значения измерений, приведённые к строковому типу; поле _measure – последний атрибут кортежа, представляющий ячейку гиперкуба.

```
class Tuple
{
public:
    Tuple(const QStringList & keys, double measure);
    const QStringList & keys() const;
    void setKeys(const QStringList & keys);
    double measure() const;
    void setMeasure(double measure);
private:
    QStringList _keys;
    double _measure;
};
```

Собственно гиперкуб является generic-контейнером QList, содержащим множество экземпляров класса Tuple, описанного в листинге 2, и представляет собой мгновенный снимок данных.

Следующая проблема – выбор осей гиперкуба для формирования отчёта и передача этой информации коду, который будет его формировать. Для пользователя естественным представлением осей гиперкуба являются два списка их названий: один для строк отчёта, второй – для столбцов, допускающие множественный выбор и перемещение элементов списка относительно друг друга. С точки зрения кода, названия измерений использовать неудобно, т.к. значения измерений хранятся в QStringList, который позволяет обращаться к элементам по индексам. Когда пользователь меняет измерения местами с целью получения в отчёте необходимой детализации и агрегации, их порядок уже не будет соответствовать тому, который был задан при написании запроса, поэтому для соответствия названий измерений их порядковым индексам предлагается использовать словарь QMap<QString, int>, в котором в качестве ключей хранятся названия измерений, в качестве значений – их порядковые индексы.

Отчёт, который строится алгоритмом, представлен в виде вектора векторов. Для удобства вектор инкапсулирован в класс Projection, который показан в листинге 3.

Листинг 3. Представление отчёта, где поле класса _data является вектором векторов: двумерным массивом, содержащим ячейки отчета.

```
class Projection
{
public:
    Projection(int width = 0, int height = 0);
    Projection(const QSize & size);
    void resize(int width, int height);
    void resize(const QSize & size);
    void clear();
    int width();
    int height();
    QVector<Cell> & operator[](int i);

private:
    QVector< QVector<Cell> > _data;
};
```

Листинг 4. Представление ячейки отчёта, где поле _value аккумулирует значения, прибавляемые к ячейке; _min, _max содержат минимальное и максимальное значения ячейки соответственно; _count считает количество слагаемых для расчёта среднего значения ячейки.

```
class Cell
{
public:
    Cell();
    Cell(int x, int y, double value);
    double value() const;
    void setValue(double value);
    int x() const;
    void setX(int x);
    int y() const;
    void setY(int y);
    double max() const;
    double min() const;
    double avg() const;
    Cell operator+=(double value);
    const Cell operator+(const Cell & other);
    bool isEmpty();
    void clear();
private:
    int _x;
    int _y;
    double _value;
    double _max;
    double _min;
    int _count;
};
```

В листинге 4 можно увидеть, что алгоритм даёт возможность выбора величин отчёта, которые будут показаны пользователю в конечном счёте: сумма, минимум, максимум, среднее. Формирование самого отчёта выполняется в три этапа:

1. Сканирование гиперкуба (generic-контейнера QList) с целью формирования словарей с полными уникальными ключами (заголовки колонок и строк отчёта);
2. Сканирование полученных словарей с целью задания монотонно возрастающих целочисленных значений с шагом 1 (номера колонок и строк отчёта по порядку);
3. Сканирование гиперкуба с целью восстановления полученных ранее ключей и получения по ним координат ячеек отчёта из сформированных словарей, заполнение отчёта;

В первых реализациях все три пункта выполнялись в одном потоке, что при наличии многоядерных процессоров являлось само по себе не эффективным. Поэтому, с целью оптимизации быстродействия алгоритма, было решено

провести распараллеливание с помощью технологии MapReduce. Предварительный анализ затрат времени показал, что на пункты 1 и 3 алгоритма приходится свыше 95% общих затрат времени, поэтому пункт 2 распараллеливать нецелесообразно.

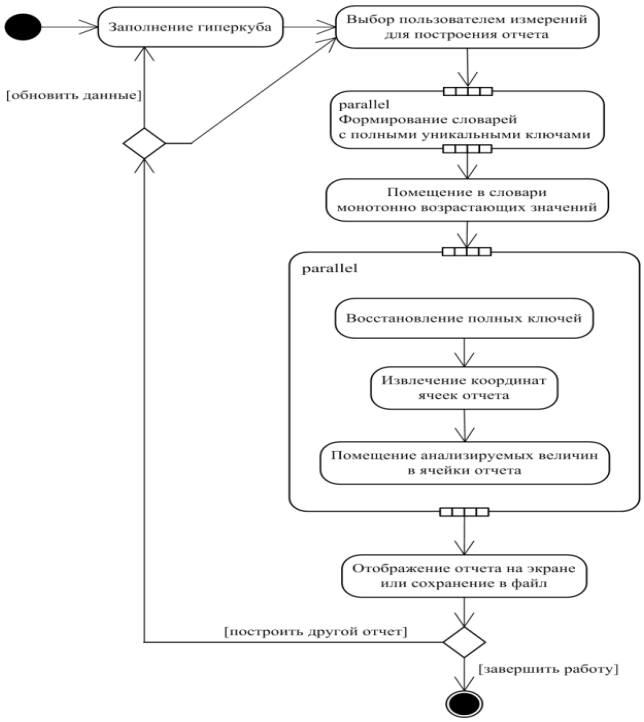

Рис. 1. Диаграмма деятельности.

Пункты 1 и 3, предложенные к распараллеливанию, не являются потоково-безопасными – в них осуществляется модификация глобальных объектов: словари с полными ключами и ячейки отчёта. Но одним из плюсов MapReduce является возможность абстрагирования от использования низкоуровневых примитивов синхронизации, поэтому приведенные ниже фрагменты кода не содержат ни мьютексов, ни семафоров, ни блокировок на чтение-запись.

Так, пункт 1 реализован следующим образом:

Листинг 5. Функции Map и Reduce для формирования словарей с полными ключами.

```
QMap<QString, int> _dimensions;
QMap<QString, int> _xCoordinates, _yCoordinates;
QStringList _xDimensions, _yDimensions;
QPair<QString, QString> mapFullKeys(const Tuple &
tuple)
{
    QPair<QString, QString> keys;
```

```
        QString key;
        int i = 0;
        foreach (const QString & dimension, _xDimensions) {
            i = _dimensions[dimension];
            key += tuple.keys()[i];
        }
        keys.first = key;
        key.clear();
        foreach (const QString & dimension, _yDimensions) {
            i = _dimensions[dimension];
            key += tuple.keys()[i];
        }
        keys.second = key;
        return keys;
    }
    void reduceFullKeys(int & /*i*/, const QPair<QString,
QString> & keys)
    {
        _xCoordinates[keys.first] = -1;
        _yCoordinates[keys.second] = -1;
    }
```

Функция mapFullKeys в листинге 5 вызывается для каждого элемента гиперкуба параллельно. Имена выбранных для отчёта измерений представлены в виде двух списков: _xDimensions, _yDimensions. С помощью словаря _dimensions восстанавливаются индексы измерений и уже по индексам извлекаются их значения. В случае если на строки отчёта отображаются несколько измерений гиперкуба, то ключ будет представлять собой конкатенацию значений измерений. То же самое справедливо и для столбцов. В итоге функция возвращает пару ключей, которая автоматически передается функции reduceFullKeys, которая, в свою очередь, помещает ключи в формируемые словари _xCoordinates и _yCoordinates. В итоге словари будут состоять из уникальных ключей, автоматически отсортированных в порядке возрастания.

На втором шаге оба словаря перебираются по порядку и ключам присваиваются возрастающие значения: 0, 1, 2, 3 и т.д. Таким образом, словари в качестве ключей содержат заголовки столбцов и строк будущего отчёта, а в качестве значений – их порядковые индексы.

Реализация третьего шага отображена ниже:

Листинг 6. Функции Map и Reduce для заполнения отчёта.

```
    QMap<QString, int> _dimensions;
    QMap<QString, int> _xCoordinates, _yCoordinates;
    QStringList _xDimensions, _yDimensions;
    Projection _projection;
    Cell mapProjection(const Tuple & tuple)
    {
```

```
    QString key;
    foreach (const QString & dimension, _xDimensions) {
        int i = _dimensions[dimension];
        key += tuple.keys()[i];
    }
    int x = _xCoordinates[key];
    key.clear();
    foreach (const QString & dimension, _yDimensions) {
        int i = _dimensions[dimension];
        key += tuple.keys()[i];
    }
    int y = _yCoordinates[key];
    return Cell(x, y, tuple.measure());
}
void reduceProjection(int & /*i*/, const Cell & cell)
{
    _projection[cell.x()][cell.y()] += cell.value();
}
```

Функция mapProjection в листинге 6 строит уникальные ключи по той же схеме, что и mapFullKeys в листинге 5. По ключам из сформированных ранее словарей _xCoordinates и _yCoordinates извлекаются координаты ячеек отчёта, а из текущего кортежа гиперкуба извлекается величина *_measure*. Все три значения передаются в функцию reduceProjection в виде экземпляра класса Cell. В свою очередь, reduceProjection по указанным координатам получает ячейку отчёта и к её значению прибавляет величину *_measure*.

После распараллеливания алгоритма была проведена экспериментальная оценка затрат времени на построение отчета по различным сочетаниям измерений гиперкуба. Для того чтобы наглядно убедиться в эффективности MapReduce, описанная выше реализация сравнивается с тестовой реализацией, выполненной по технологии PTHREAD[10] с использованием примитивов синхронизации.

Априори известно, что затраты времени зависят от ряда как контролируемых, так и неконтролируемых факторов, таких как:

1. объём выборки из базы данных, т. е. размер гиперкуба;
2. количество и длины полных ключей измерений гиперкуба, выбранных для построения отчёта;
3. количество параллельных потоков;
4. тип процессора: архитектура, количество ядер, частота, объём кэш;
5. объём и частота основной памяти;
6. тип, разрядность и версия операционной системы, частота системного таймера;
7. версия Qt Framework.

Из перечисленных факторов, факторы с 4 по 7 являются неконтролируемыми и, более того, все они быстро меняются с течением времени, поэтому их влияние можно оценить только качественно.

Первый фактор, пока объём выборки не превышает объем доступной оперативной памяти, соответствует линейной зависимости затрат времени от размера выборки; при этом соотношение затрат времени для различных вариантов реализации параллельного кода должно оставаться постоянным, поэтому в эксперименте рассматривается достаточно малый объём данных: тестовый гиперкуб имеет 9 измерений и состоит из 66 354 кортежей.

Для оценки количества и размеров составных ключей предлагается использовать обобщённый параметр K, для определения которого используется следующее соотношение, основанное на том, что среднее время поиска в дереве пропорционально его глубине, а среднее время сравнения двух строк пропорционально их длине[11]:

$$K = \log_2(N) * L \,. \qquad \textbf{(1)}$$

где: K– обобщённый параметр; N– количество ключей; L– средняя длина ключа.

Так как отчёт является двумерным массивом, то результирующий показатель будет равен сумме показателей для колонок и строк:

$$K = K_x + K_y = \log_2(X) * L_x + \log_2(Y) * L_y \,. \qquad \textbf{(2)}$$

где: K – результирующий параметр; $K_{x,y}$ – обобщённые параметры для колонок и строк; X, Y – количество ключей; $L_{x,y}$ – средние длины ключей.

Для проведения экспериментов использовался компьютер с 6-ядерным процессором AMD Phenom II X6 1100T и 4 Гб ОЗУ; операционная система Linux KUbuntu 12.04 LTS 64-bit с версией ядра 3.2; Qt Framework версии 4.8.4. Данные снимались в 10-кратной повторности. Для определения затрат времени на формирование словарей с ключами и заполнение отчёта использовался системный вызов *clock_gettime*.

На рисунке 2 представлены общие затраты времени для сравниваемых технологий: однопоточного варианта, многопоточного варианта на MapReduce и многопоточных вариантов на PTHREAD - в зависимости от значений предлагаемого обобщенного параметра K.

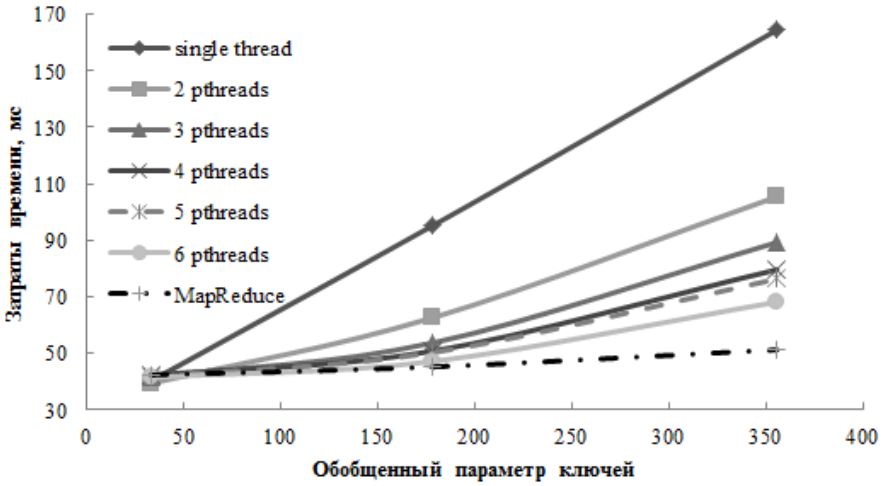

Рис. 2. Затраты времени на формирование ключей и заполнение отчета в зависимости от обобщенного параметра для 6-ядерного процессора AMD Phenom II X6 1100T, частота ядер 3.3 ГГц, кэш 6144 Кб, 64-разрядная ОС Linux 3.2, Qt 4.8.4.

Из графиков видно, что гипотеза о линейной зависимости затрат времени от обобщённого параметра K подтвердилась. Общие затраты времени для варианта с 6 потоками PTHREAD и для варианта с использованием MapReduce очень близки между собой. Однако с увеличением значения обобщенного параметра, MapReduce демонстрирует лучшую эффективность. Поскольку для конечного пользователя, QA-специалиста, важны общие затраты времени, а для разработчиков - кроссплатформенность и масштабируемость, MapReduce является верным выбором при оптимизации задачи многофакторного анализа на стороне клиента.

4 Заключение

Описанная методика анализа данных, обладая высокой эффективностью, кроссплатформенностью и масштабируемостью, является подходящей альтернативой существующим решениям при обработке результатов нагрузочного тестирования трейдинговых платформ. Алгоритм прост в использовании, т.к. его конфигурация заключается лишь в написании стандартного SQL-запроса выборки SELECT, придерживаясь указанных правил; а применение технологии MapReduce позволяет использовать доступную мощь многоядерных процессоров, которые в настоящее время устанавливаются на абсолютное большинство компьютеров. Данное решение апробируется в ряде проектов по нагрузочному тестированию компании «Exactpro Systems», LLC.

Литература

1. Иткин И.Л.: Тестирование биржевых систем в условиях высокочастотного трейдинга // SQA Days #10: [Электронный ресурс]. Режим доступа: http://sqadays.com/talk.sdf/sqadays/11151/talks/12196
2. Каширин И.Ю., Семченков С.Ю.: Интерактивная аналитическая обработка данных в современных OLAP-системах // Бизнес-информатика. 2009. Вып. 2(8).
3. SQL Server Analysis Services - Multidimensional Data // MSDN: [Электронный ресурс]. Режим доступа: http://msdn.microsoft.com/en-us/library/bb522607(v=sql.105).aspx
4. Oracle OLAP // Oracle: [Электронный ресурс]. Режим доступа: http://www.oracle.com/technetwork/database/options/olap/index.html
5. Palo by Jedox // Palo: [Электронный ресурс]. Режим доступа: http://palo.net/
6. Comparison - Palo (Open Source) / Jedox (Premium) // Palo: [Электронный ресурс]. Режим доступа: http://www.palo.net/index.php?id=13
7. Jelen Bill, Alexander Mike. Pivot table data crunching. Pearson Education: 2011.
8. Alexey Zverev: Technical Testing // EXTENT February 2011: [Электронный ресурс]. Режим доступа: http://www.slideshare.net/IosifItkin/technical-testing-introduction
9. Qt 4.8: Concurrent Programming // Qt Reference Documentation: [Электронный ресурс]. Режим доступа: http://doc-snapshot.qt-project.org/4.8/threads-qtconcurrent.html
10. Blaise Barney: POSIX Threads Programming // Lawrence Livermore National Laboratory: [Электронный ресурс]. Режим доступа: https://computing.llnl.gov/tutorials/pthreads/
11. Кнут Д. Э.: Искусство программирования, том 3. Сортировка и поиск, 2-е издание. – М.: Издательский дом «Вильямс», 2003.