**ОГЛАВЛЕНИЕ**

[Введение 6](#_Toc438595289)

[1 Понятие колоночной базы данных 7](#_Toc438595290)

[2 Получение данных мониторинга 9](#_Toc438595291)

[3 Выбор колоночных баз данных 11](#_Toc438595292)

[4 Сравнение скорости работы 12](#_Toc438595293)

[Заключение 13](#_Toc438595294)

[Список литературы 14](#_Toc438595295)

[Приложения 15](#_Toc438595296)

**РЕФЕРАТ**

Курсовой проект, 17 с., 6 источник, 3 рисунка, 2 приложение.

КОЛОНОЧНЫЕ БАЗЫ ДАННЫХ, СЧЕТЧИК ПРОИЗВОДИТЕЛЬНОСТИ, ИНТЕРФЕЙС ПРОГРАММИРОВАНИЯ ПРИЛОЖЕНИЙ.

**Объект исследования** – данные мониторинга компьютера.

**Цель работы** – определить, какая из выбранных колоночных СУБД лучше подходит для хранения данных мониторинга.

**Методы исследования** – измерительно-сравнительный эксперимент.

**Результаты** – написана программа, позволяющая получить данные мониторинга. Написаны программы, позволяющие работать с колоночными СУБД и измерять время работы запросов. Выявлена база данных, которая лучше остальных подходит для хранения данных мониторинга.

**РЭФЕРАТ**

Курсавы праект, 17 с., 6 крыніца, 3 малюнка, 2 дадатка.

СЛУПКОВЫЯ БАЗЫ ДАНЫХ, ЛІЧЫЛЬНІК ПРАДУКЦЫЙНАСЦІ, ІНТЭРФЕЙС ПРАГРАМАВАННЯ ПРЫКЛАДАННЯЎ.

**Аб'ект даследавання** − даныя маніторынгу кампутара.

**Мэта працы** − вызначыць, якая з выбраных слупковых СКБД лепш падыходзіць для захоўвання даных маніторынгу.

**Метады даследавання** - вымяральна-параўнальны эксперымент.

**Вынікі** − напісана праграма, якая дазваляе атрымаць даныя маніторынгу. Напісаныя праграмы, якія дазваляюць працаваць з слупковымі СКБД і вымяраць час работы запытаў. Выяўлена база даных, якая лепш астатніх падыходзіць для захоўвання даных маніторынгу.

**ABSTRACT**

A course project, 17 p., 6 sources, 3 figures, 2 additions.

COLUMNAR DATABASE, PERFOMANCE COUNTER, APPLICATION PROGRAMMING INTERFACE.

**Object** – computer monitoring data.

**Purpose** – to determine which of the selected columnar DBMS better suited for storing monitoring data.

**Methods** – measuring and comparative experiment.

**Results** – written program that allows you to obtain monitoring data. Written a program that can handle columnar databases and measure the elapsed time for requests. Revealed database that are better than others in storing monitoring data.

# **Введение**

Колоночные СУБД призваны решить проблему неэффективной работы традиционных СУБД в аналитических системах и в системах с подавляющим большинством операций типа «чтение». Они позволяют на более дешевом и маломощном оборудовании получить прирост скорости выполнения запросов в 5, 10 и иногда даже в 100 раз, при этом, благодаря компрессии, данные будут занимать на диске в 5-10 раз меньше, чем в случае с традиционными СУБД.

У колоночных СУБД есть и недостатки — они медленно работают на запись, не подходят для транзакционных систем и, как правило, ввиду «молодости» имеют ряд ограничений для разработчика, привыкшего к развитым традиционным СУБД.

Колоночные СУБД применяются, как правило, в аналитических системах класса business intelligence и аналитических хранилищах данных.[1.]

В данном курсовом проекте будет сравнение различных колоночных СУБД при хранении данных производительности компьютера.

Для получения данных мониторинга используется программа на языке C++. Для сравнения различных колоночных СУБД используются разные программы на языках C++ и Javascript.

# **Понятие колоночной базы данных**

Для получения представления о принципе поколоночного хранения данных рассмотрим принцип хранения в традиционных строчных базах данных.

Под построчным хранением данных обычно понимается физическое хранение всей строки таблицы в виде одной записи, в которой поля идут последовательно одно за другим, а за последним полем записи в общем случае идет первое следующей записи. Приблизительно так:

[A1, B1, C1], [A2, B2, C2], [A3, B3, C3]...

где A, B и С — это поля (столбцы), а 1, 2 и 3 — номер записи (строки).

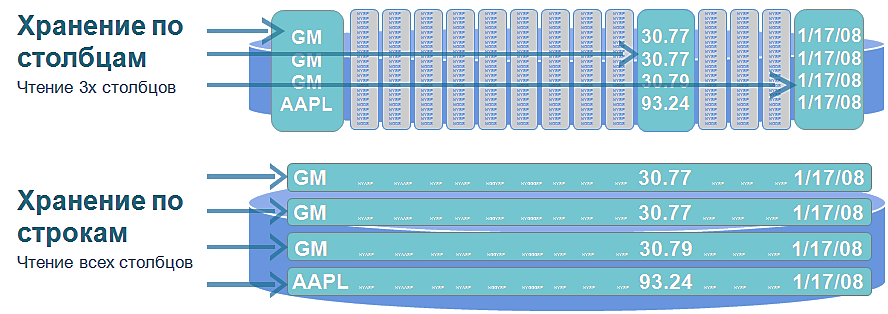
Такое хранение чрезвычайно удобно для частых операций добавления новых строк в базу данных, хранящуюся, как правило, на жестком диске – ведь в этом случае новая запись может быть добавлена целиком всего за один проход головки накопителя. Существенные ограничения по скорости, накладываемые накопителем на жёстких магнитных дисках, вызвали также необходимость ведения специальных индексов, которые позволяли бы отыскивать нужную запись на диске за минимальное количество проходов головки HDD. Обычно формируется несколько индексов, в зависимости от того, по каким полям требуется делать поиск, что увеличивает объем базы данных на диске иногда в несколько раз. Для отказоустойчивости, традиционные СУБД автоматически дублируют операции в логах, что приводит к еще большему месту, занимаемому на дисках. В итоге, например среднестатистическая база данных Oracle занимает на диске в 5 раз больше места, чем объем полезных данных в ней.

Однако что произойдет, если выбрать, например, только 3 поля из таблицы, в которой их всего 50? В силу построчного хранения данных в традиционных СУБД будут прочитаны абсолютно все строки целиком со всеми полями. Это значит, что не важно, нужны ли нам только 3 поля или 50, с диска в любом случае они все будут прочитаны целиком и полностью, пропущены через контроллер дискового ввода-вывода и переданы процессору, который уже отберет только необходимые для запроса. К сожалению, каналы дискового ввода-вывода обычно являются основным ограничителем производительности аналитических систем. Как результат, эффективность традиционной СУБД при выполнении данного запроса может снизиться в 10–15 раз из-за неминуемого чтения лишних данных. Причем действие закона Мура на скорость ввода-вывода дисковых накопителей куда слабее, чем на скорость процессоров и объемы памяти. Так что дальше ситуация будет только усугубляться.

Решить эту проблему призваны колоночные СУБД. Основная идея колоночных СУБД — это хранение данных не по строкам, как это делают традиционные СУБД, а по колонкам. Это означает, что с точки зрения SQL-клиента данные представлены как обычно в виде таблиц, но физически эти таблицы являются совокупностью колонок, каждая из которых, по сути, представляет собой таблицу из одного поля. При этом физически на диске значения одного поля хранятся последовательно друг за другом — приблизительно так:

[A1, A2, A3], [B1, B2, B3], [C1, C2, C3] и т.д.

Такая организация данных приводит к тому, что при выполнении select в котором фигурируют только 3 поля из 50 полей таблицы, с диска физически будут прочитаны только 3 колонки. Это означает, что нагрузка на канал ввода-вывода будет приблизительно в 50/3=17 раз меньше чем при выполнении такого же запроса в традиционной СУБД (рис. 1).



**Рисунок 1**− Операция чтения в колоночных и строковых СУБД

Кроме того, при поколоночном хранении данных появляется замечательная возможность сильно компрессировать данные, так как в одной колонке таблицы данные, как правило, однотипные, чего не скажешь о строке. Алгоритмы компрессии могут быть разные. Приведем пример одного из них — так называемого Run-Length Encoding (RLE):

Если у нас есть таблица со 100 млн записей, сделанных в течение одного года, то в колонке «Дата» на самом деле будет храниться не более 366 возможных значений, так как в году не более 366 дней. Поэтому мы можем 100 млн отсортированных значений в этом поле заменить на 366 пар значений вида <дата, количество раз> и хранить их на диске в таком виде. При этом они будут занимать приблизительно в 100 тыс. раз меньше места, что также способствует повышению скорости выполнения запросов.

[1.]

# **Получение данных мониторинга**

Для данного курсового проекта будем анализировать следующие данные мониторинга:

* производительность процессора (% активного времени),
* количество выделенной логической памяти (% использованных байт),
* активное время физического жесткого диска (% активного времени).

Количество данных каждого типа: 10000.

Для получения данных мониторинга напишем программу на языке C++, которая будет получать нужные характеристики с частотой 1 секунда и записывать их в разные текстовые файлы (см. Приложение А).

В данной программе используем привязку языка C++ к среде программирования .NET, которая называется C++/CLI. Для этого в Microsoft Visual Studio в настройках проекта во вкладке «Общие» полю «Поддержка общеязыковой среды выполнения (CLR)» присваиваем значение «Поддержка CLR-среды (/clr)».

Схема работы программы представлена на рис. 2

TimerObject

PerfomanceCounter

Processor

PerfomanceCounter

Memory

PerfomanceCounter

Disk

Get

Get

Get

Writer

Processor

Writer

Memory

Writer

Disk

**Рисунок 2**− Принцип работы программы

В начале создаем потоки вывода. Далее специальные строки, которые соответствуют конкретным названиям категорий и счетчиков. Потом проверяем, правильно ли мы указали эти категории и счетчики.

Присваиваем статические поля объекту TimerObject. Создаем новый таймер и запускаем. Работать он будет с частотой 1 секунда и на каждой итерации обнуляться.

На каждом цикле TimerObject записывает в разные файлы данные, которые возвращают счетчики.

# **Выбор колоночных баз данных**

Для данной курсовой работы будут использованы следующие колоночные базы данных:

* Apache CouchDB [2.],
* Kx analytics database [3.],
* LucidDB [4.],
* Sybase IQ [5.].

Во всех СУБД используются свои API, поэтому каждую программу для доступа к конкретной базе данных удобнее делать отдельно.

Для доступа к Apache CouchDB будем использовать программу на Javascript, т.к. эта СУБД обрабатывает HTTP запросы, с данными в формате JSON.

Для остальных СУБД можно написать программы на C++, использующие свои интерфейсы программирования приложений (API).

Структура всех приложений одинакова (рис. 3)

Обработчик

Входной файл

Processor

Входной файл

Memory

Входной файл

Disk

Запись в базу данных

Processor

Запись в базу данных

Memory

Запись в базу данных

Disk

**Рис 3**− Принцип работы приложений

Приложения считывают файлы о производительности, записанные ранее, и через API записывают в базы данных эту информацию, причем делают это одной командой для лучшего быстродействия.

# **Сравнение скорости работы**

При записи в базы, считаем затраченное время. Важно отметить, что нужно исключить время чтения файлов мониторинга.

Далее узнаем время запроса вида (по аналогии с SQL):

SELECT processor, disk FROM table

Для лучшей оценки сделаем 3 запроса и найдем среднее.

Результаты приведены в таблице (см. Приложение Б).

Как видим, скорость работы разных СУБД не сильно отличается, однако, все же, можно выделить «победителей» и «проигравших».

Наихудший результат по всем параметрам показала СУБД LucidDB.

Наивысшая скорость записи у Apache CouchDB, в то же время как наивысшая скорость чтения у Kx analytics database.

Стоит заметить, что, несмотря на то, что скорость записи Apache CouchDB выше, нужно признать, что «победитель соревнований» − Kx analytics database, т.к. для анализа таких данных, как производительность компьютера, значительно чаще выполняются запросы чтения.

# **Заключение**

В данном курсовом проекте рассмотрено понятие колоночной базы данных, рассмотрены различные колоночные СУБД.

Были написаны программы для мониторинга компьютера и записи данных в СУБД.

Было измерено время обработки запросов чтения и записи разных СУБД. На основании времен обработки была выбрана СУБД, которая лучше остальных подходит для хранения данных мониторинга.

# **Список литературы**

1. Колоночные СУБД — принцип действия, преимущества и область применения [http://habrahabr.ru/post/95181/]
2. Apache CouchDB [http://couchdb.apache.org/]
3. Kx analytics database [http://kx.com/documentation.php]
4. LucidDB [http://luciddb.sourceforge.net/]
5. Sybase CIS|IQ [https://www.sybase.ru/products/sybase\_iq]
6. The Top Column-Oriented Databases Compared [http://www.timestored.com/time-series-data/column-oriented-databases]

# **Приложения**

**Приложения А**

**Текст программы**

#using <system.dll>

using namespace System;

using namespace System::Threading;

using namespace System::Diagnostics;

using namespace System::Timers;

using namespace System::IO;

ref struct TimerObject

{

public:

static StreamWriter^ swProcessor;

static StreamWriter^ swMemory;

static StreamWriter^ swDisk;

static String^ m\_instanceName;

static PerformanceCounter^ m\_processorCounter;

static PerformanceCounter^ m\_memoryCounter;

static PerformanceCounter^ m\_diskCounter;

static void OnTimer(Object^ source, ElapsedEventArgs^ e)

{

try

{

swProcessor->WriteLine(m\_processorCounter->NextValue( ).ToString("f"));

swMemory->WriteLine(m\_memoryCounter->NextValue( ).ToString("f"));

swDisk->WriteLine(m\_diskCounter->NextValue( ).ToString("f"));

}

catch(Exception^ e)

{

if (dynamic\_cast<InvalidOperationException^>(e))

{

Console::WriteLine("Instance '{0}' does not exist",

m\_instanceName);

return;

}

else

{

Console::WriteLine("Unknown exception... ('q' to quit)");

return;

}

}

}

};

int main()

{

StreamWriter^ swProcessor = gcnew StreamWriter("processor.txt");

StreamWriter^ swMemory = gcnew StreamWriter("memory.txt");

StreamWriter^ swDisk = gcnew StreamWriter("disk.txt");

String^ instanceName = "\_Total";

String^ objectProcessor = "Processor";

String^ counterProcessor = "% Processor Time";

String^ objectMemory = "Memory";

String^ counterMemory = "% Committed Bytes In Use";

String^ objectDisk = "PhysicalDisk";

String^ counterDisk = "% Disk Time";

try

{

if ( !PerformanceCounterCategory::Exists(objectProcessor) || !PerformanceCounterCategory::Exists(objectMemory) || !PerformanceCounterCategory::Exists(objectDisk) )

{

Console::WriteLine("Object does not exist");

return -1;

}

}

catch (UnauthorizedAccessException ^ex)

{

Console::WriteLine(ex->Message);

return -1;

}

if ( !PerformanceCounterCategory::CounterExists(counterProcessor, objectProcessor)

|| !PerformanceCounterCategory::CounterExists(counterMemory, objectMemory)

|| !PerformanceCounterCategory::CounterExists(counterDisk, objectDisk) )

{

Console::WriteLine("Counter does not exist");

return -1;

}

TimerObject::swProcessor = swProcessor;

TimerObject::swMemory = swMemory;

TimerObject::swDisk = swDisk;

TimerObject::m\_instanceName = instanceName;

TimerObject::m\_processorCounter = gcnew PerformanceCounter(objectProcessor, counterProcessor, instanceName);

TimerObject::m\_memoryCounter = gcnew PerformanceCounter(objectMemory, counterMemory);

TimerObject::m\_diskCounter = gcnew PerformanceCounter(objectDisk, counterDisk, instanceName);

System::Timers::Timer^ aTimer = gcnew System::Timers::Timer();

aTimer->Elapsed += gcnew ElapsedEventHandler(&TimerObject::OnTimer);

aTimer->Interval = 1000;

aTimer->Enabled = true;

aTimer->AutoReset = true;

Thread::Sleep(10000000);

swProcessor->Close();

swMemory->Close();

swDisk->Close();

return 0;

}

**Приложение Б**

**Таблица результатов запросов**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Запросы  СУБД | Время записи, мс | Время чтения, мс | | | |
| Запрос 1 | Запрос 2 | Запрос 3 | Среднее |
| Apache CouchDB | 943 | 14 | 14 | 13 | 13,3 |
| Kx analytics database | 978 | 12 | 13 | 12 | 12,3 |
| LucidDB | 1032 | 14 | 14 | 16 | 14,7 |
| Sybase IQ | 990 | 15 | 11 | 12 | 12,7 |