УДК 004.043, 004.65

ОБЗОР УСЛОВИЙ АДАПТАЦИИ САМОАДАПТИРУЮЩИХСЯ АССОЦИАТИВНЫХ КОНТЕЙНЕРОВ ДАННЫХ

Д. Р. Потапов, М. А. Артемов, Е. С. Барановский

Воронежский государственный университет

Поступила в редакцию 07.02.2017 г.

Аннотация. В статье рассматривается проблема построения самоадаптирующихся ассоциативных контейнеров данных. Эта проблема актуальна в связи с наличием большого количества контейнеров и необходимостью выбора оптимального контейнера для каждого конкретного случая. В работе представлен обзор и анализ условий, в которых происходит адаптация контейнеров, таких как: объем информации, количество запросов и архитектура информационной системы. Кроме того, выявлены основные ограничения, которые необходимо учитывать при реализации самоадаптирующихся ассоциативных контейнеров данных.
Ключевые слова: хранение информации, хранилище данных, оптимальный контейнер, адаптивный контейнер, условия адаптации.
Annotation. The work is dedicated to the question of adaptive associative data storage building, which is actual because of existing the large number of storages and necessity to select the best optimal storage in each case. Adaptation conditions, such as information volume, amount of requests and information system architecture are also analyzed and reviewed in this paper. In this work the main constraints for adaptive storage implementation are also identified.
Keywords: store the data, data storage, optimal container, adaptive container, adaptation conditions.

1. ВВЕДЕНИЕ

В любой задаче обработки данных явно или неявно требуются эффективно работающие контейнеры данных [1]. Количество типов таких контейнеров огромно, но каждый из них удобен только для ограниченного количества задач. При этом зачастую условия задачи меняются в процессе работы и, следовательно, для максимально эффективной работы периодически необходима смена типа контейнера.

Таким образом, возникает необходимость создания самоадаптирующегося ассоциативного контейнера данных, который меняет логику своей работы в зависимости от нагрузки. Например, при большом количестве операций поиска контейнер работает на основе одной структуры данных, при большом количестве вставок – на основе другой, при определенном соотношении – на основе третьей. Помимо обеспечения максимальной скорости работы должна использоваться та структура данных, которая использует как можно меньше дополнительной памяти.

Одним из очевидных случаев использования таких контейнеров являются мобильные устройства, такие как планшетные ПК, смартфоны, мобильные телефоны, ноутбуки и др. Благодаря развитию технологий такие устройства имеют весьма компактные размеры и отличную автономность, что позволяет их легко транспортировать. Тем не менее, мобильные устройства в силу своих размеров все равно имеют ограниченные ресурсы, которые существенно уступают стационарным компьютерам [2]. Ограниченными являются в первую очередь оперативная память и мощность процессора.

В этом случае и появляется необходимость в адаптивных контейнерах данных. Они позволяют максимально эффективно использовать ресурсы устройства. При изме-

няющейся нагрузке выбирается контейнер, оптимальный либо с точки зрения быстродействия, либо – минимальных затрат оперативной памяти.

Настоящее исследование является актуальным, так как существует большое количество контейнеров и их комбинаций, при этом необходимо динамически выбирать из них оптимальное решение для каждого конкретного случая на основе различных критериев, например, статистики запросов к контейнеру данных или объема занимаемой памяти.

Целью данной работы является рассмотрение задачи построения самоадаптирующихся ассоциативных контейнеров данных и выявление основных ограничений, которые необходимо учитывать при их реализации.

Обзор условий адаптации. Для того чтобы построить самоадаптирующийся ассоциативный контейнер данных, необходимо понимать, в каких условиях происходит адаптация. Нужно знать, какие ограничения могут накладывать задачи, для которых используются эти контейнеры. Среди основных условий можно выделить следующие:

1. объем информации;
2. количество и типы запросов;
3. архитектура информационной системы.

2. ОБЪЕМ ИНФОРМАЦИИ

Объем информации является основным условием адаптации контейнера, так как это условие определяет, где располагается контейнер. Существует несколько вариантов его расположения (в порядке увеличения объёма информации):

1. данные располагаются в памяти;
2. данные располагаются частично в памяти, частично на диске;
3. данные располагаются на кластере.

2.1. Данные в памяти

В первую очередь необходимо понимать, что контейнер использует несколько уровней памяти компьютера [3]. Верхний уровень иерархии памяти – это кэш процессора. Здесь можно использовать особенности кэша. Как известно, современные процессоры обращаются к памяти небольшими блоками (обычно 64 байта), а не по одному байту, как было в первых моделях компьютеров. Эти блоки называются строками кэша. Таким образом, при извлечении какого-либо значения из памяти в кэше окажется как минимум одна его строка. При последующих обращениях к любому значению из данной строки доступ будет происходить достаточно быстро. При этом от расположения данных в памяти зависит, сколько раз к ней будут обращаться. Например, в случае если данные не выровнены, может потребоваться два запроса к оперативной памяти вместо одного. Кроме того, обычно современные процессоры имеют несколько (два или три) уровней кэша (они называются L1, L2 и L3), что ведет к падению производительности при достижении соответствующих порогов кэшей. Следовательно, на этом уровне необходимо учитывать размер кэшей и при необходимости выравнивать данные.

Следующий уровень иерархии памяти – это оперативная память [4]. В данном случае контейнер помещается в размеры оперативной памяти и, следовательно, не требуется дополнительных операций для хранения информации. При перестроении контейнера в оперативной памяти не требуется учитывать такие дополнительные факторы, как, например, затраты на доступ к диску или передача данных по сети (которые являются гораздо более существенными, чем затраты на многопоточность и время ожидания блокировок). Кроме того, не нужны специальные алгоритмы для уменьшения затрат на вышеуказанные проблемы. Таким образом, для контейнера можно использовать любые подходящие для данной нагрузки структуры данных.

Помимо иерархии памяти при проектировании контейнера следует учитывать ещё и архитектуру соединения процессоров с оперативной памятью. Существует три архитектуры: Uniform Memory Access, Massive Parallel Processing, Non-Uniform Memory Access [5].

Uniform Memory Access (однородный доступ к памяти) – архитектура, в которой используется общая системная память. Все процессоры подключены к ней с помощью шины. Данная архитектура также называется SMP

(Symmetric Multiprocessing – симметричная многопроцессорная обработка), потому что процессоры подключены к памяти симметрично и имеют к ней равный однородный доступ. Однородный доступ к памяти означает, что все процессоры обращаются к памяти единообразно, имеют равные права на доступ, и доступ осуществляется за одинаковое время.

Использование SMP не требует специальных моделей программирования при создании приложений, однако обычно используется модель параллельных ветвей. Все процессоры в данной модели работают независимо друг от друга. Из этого следует, что проектирование адаптивного контейнера для этой архитектуры требует решения проблемы многопоточности.

Другой архитектурой соединения процессоров с оперативной памятью является MPP (Massive Parallel Processing). В данной архитектуре вся система делится на узлы, в каждом из которых процессоры ограничены локальными ресурсами. При этом обмена данными между узлами в данной архитектуре не предусмотрено. Для работы с MPP используется специальная парадигма программирования с передачей данных Massive Passing Programming Paradigm – (MPI, PVM, BSPlib) [6], т. е. при разработке адаптивного контейнера необходимо самостоятельно реализовывать коммуникации, распределение и планировку задач на узлах. Кроме того, при перестраивании контейнера также необходимы дополнительные алгоритмы [7], так как разделяемая между процессорами память отсутствует, и каждый процесс ограничен объемом локальной памяти.

Предыдущие перечисленные архитектуры имеют существенные недостатки. Поэтому была создана другая архитектура – Non-Uniform Memory Access (NUMA), которая наследует от Uniform Memory Access и Massive Parallel Processing лучшие черты. В данной архитектуре система делится на узлы, каждый из которых помимо доступа к локальным ресурсам имеет ещё и доступ к удаленной памяти, т. е. памяти других узлов. Естественно, время доступа к памяти других узлов гораздо больше, чем к памяти своего узла. Для использования этой архитектуры применяется гибридная парадигма программирования, которая сочетает многопоточную модель SMP и многозадачную MPP. Основная задача этой парадигмы: потоки/задачи должны максимально использовать локальную память и минимально – удаленную. В общем случае не всегда возможно сделать это рационально. Следовательно, процесс инициализации данных (т. е. какие данные закрепляются за каким узлами) требует особого внимания.

Одним из примеров сложности использования NUMA является выделение памяти на различных операционных системах. Одна из команд выделения памяти – malloc. В операционной системе Linux [8] данная команда лишь резервирует память, и лишь при фактическом обращении к ней происходит физическое выделение. Таким образом можно эффективно применять NUMA для данной операционной системы, так как память будет выделяться автоматически на том же узле, который будет ее использовать.

В другой операционной системе – Windows команда malloc работает по-другому. В Windows память физически выделяется при вызове функции, т. е. на узле, который вызывает malloc. Таким образом, память может получиться удаленной для потоков, которые ее активно используют. Но в Windows существует и другая команда – VirtualAlloc, которая хорошо подходит для использования в NUMA (она работает аналогично команде malloc в Windows). Кроме того, существует специальная команда VirtualAllocExNuma из Windows NUMA API [9].

Из всего вышеперечисленного следует, что использование NUMA при проектировании адаптивного контейнера может добавить производительности, но при разработке необходимо учитывать большое количество особенностей NUMA для извлечения максимальной выгоды.

2.2. Данные частично в памяти, частично на диске

На практике количество данных может значительно превышать объем оперативной

памяти, и поэтому необходимо использовать дополнительную память. В этом случае наиболее часто используются следующие накопители: HDD и SSD – накопитель на базе технологии флэш-памяти [10].

Для HDD диска время доступа к оперативной памяти в 10^4 раз меньше, чем к жесткому диску, а для последних моделей SSD – в десятки раз меньше.

При сравнении по этой характеристике SSD выглядит значительно производительней для работы с данными. Однако коэффициент неисправимых битовых ошибок у SSD больше, чем HDD, т. е. у них выше вероятность потери данных. Следовательно, SSD стоит использовать, учитывая этот фактор. Например, его можно использовать как кэш для базы данных (например, Buffer Pool Extension в SQL Server) [11].

Тем не менее, для обоих типов накопителей любая операция чтения или записи замедляет контейнер в тысячи (для SSD – в десятки) раз, и необходимо использовать другие алгоритмы и структуры данных. Основная идея этих алгоритмов заключается в кэшировании данных.

В качестве простейшего примера изменения алгоритма можно взять стек. В случае с оперативной памятью это обычный массив в памяти, который имеет две операции: добавить и извлечь из стека. При этом все элементы массива хранятся в памяти и в любой момент к ним можно обратиться, не сильно обращая внимания на то, сколько времени займет доступ. В случае с жестким диском такой алгоритм нельзя использовать, так как доступ к каждому элементу отдельно неоправданно долог. Для того чтобы этого избежать, часть стека, в которую поступает и из которой извлекается элемент (последний блок некоторой длины), должна находиться в оперативной памяти для быстрого доступа. Все операции вставки и извлечения при этом происходят очень быстро, так как находятся в оперативной памяти, и сохраняются на жесткий диск только после того, как блок закончится.

Более того, чтобы избежать ситуации, когда элемент извлекается и необходимо подгрузить блок с жесткого диска, а потом опять добавляется, и этот блок необходимо записать на диск, в оперативной памяти обычно хранятся два блока; и предпоследний из них записывается на диск только после заполнения обоих блоков.

Этот пример показывает, что для всех возможных структур данных [12] необходимо учитывать работу с диском. Некоторые из них изначально эффективно работают на жестком диске (B-дерево, B*-дерево, B+-дерево, хеш-таблица, Bitmap), для других необходимо изменять алгоритмы для уменьшения количества обращений к диску. Данный факт сильно влияет на выбор оптимального контейнера и является определяющим для большого количества данных.

2.3. Данные располагаются на кластере

В случае, когда количество данных огромно (десятки и сотни террабайт) и дискового пространства не хватает или необходимо параллельно обрабатывать [13] эти данные, используется компьютерный кластер [14]. Для выбора оптимального контейнера в случае кластера необходимо учитывать, как организовано хранение данных на нескольких связанных компьютерах [15], [16].

На данный момент наиболее распространенным видом хранения информации являются распределенные файловые системы. В последнее время на практике часто применяются Google File System от Google [17] и Hadoop Distributed File System от Apache Software Foundation [18]. С точки зрения пользователя эти системы почти не отличаются от обычных файловых систем за исключением операции обновления файлов. В этом случае данные хранятся на различных узлах, и поэтому файловая система организована другим образом. Главное отличие файловой системы кластера от диска в том, что все коммуникации между клиентом и сервером имен/серверами данных обычно происходят по сети, и следовательно все операции чтения/записи данных в кластере ограничиваются максимальной скоростью передачи по сети.

Таким образом, алгоритм выбора оптимального контейнера должен учитывать, что

любое обращение к данным невероятно дорого с точки зрения затрат системы и производительности. Кроме того, оптимальный контейнер предполагает перестроение контейнера данных, что влечет за собой еще большие нагрузки на сеть и файловую систему.

Исходя из всего вышеперечисленного, можно сказать, что применение оптимального контейнера данных в условиях распределенной файловой системы требует очень сложного и продуманного алгоритма [19] построения и перестроения контейнера данных. При этом в большинстве случаев выгода от использования оптимального контейнера нивелирует затраты только при долгом использовании одной модели нагрузки и не частой смене контейнеров.

Альтернативой распределенным файловым системам является шардинг [20]. Шардинг – это техника масштабирования приложений, которая позволяет разделять данные между различными физическими серверами. Суть шардинга заключается в разнесении данных по разным шардам на основе некоторого ключа шардинга. Сущности, которые имеют одинаковый ключ шардинга, объединяются в одну группу данных по заданному ключу, и хранится эта группа на одном физическом шарде.

Использование этой техники существенно облегчает обработку данных. В случае с оптимальным контейнером шардинг сложно использовать с извлечением большой выгоды по производительности, так как для шардинга необходимо выбрать определенную функцию, которая зависит от структуры данных. То есть при адаптации контейнера в большинстве случаев критерий шардинга необходимо изменить для его эффективного использования (в этом случае потребуется много ресурсов на решардинг), либо оставить критерий шардинга таким же (в этом случае количество данных на узлах весьма неравномерно, что может привести к большой потере производительности).

Другой техникой масштабирования является репликация [20], [21]. Суть данного приёма заключается в непрерывном копировании (реплицировании) с одного сервера данных на другие, которые называются репликами. Таким образом, приложение может использовать для обработки всех запросов не один сервер, а несколько. Следовательно, при использовании репликации нагрузку можно распределить с одного сервера на несколько серверов.

Существует асинхронная и синхронная репликация. При использовании асинхронной репликации обновление данных на одной реплике попадает на другие не в одной транзакции, а спустя некоторое время. Таким образом, в случае асинхронной репликации реплики могут отличаться друг от друга в течение некоторого времени (времени ожидания).

Для большинства возможных применений адаптивного контейнера такая репликация недопустима, поскольку отсутствуют гарантии, что после записи элемента в контейнер он будет доступен на любом сервере в любой момент времени. Использование этого типа репликации возможно только для контейнеров, используемых в приложениях, где актуальность данных не так критична, как производительность (за счет уменьшения сетевого трафика).

При использовании синхронной репликации обновление данных на одной реплике попадает на другие в одной транзакции, что означает, что данные логически всегда идентичны на репликах. Использование этого типа репликации лучше подходит для контейнеров, так как данные в нем всегда актуальны. Правда и у этого типа есть недостатки – при его использовании возникает необходимость обмена подтверждениями, что оказывает дополнительную нагрузку на сеть и снижается производительность, поскольку для синхронизации всех узлов необходимо некоторое время. Следовательно, выбор между синхронной и асинхронной репликацией для адаптивного контейнера должен совершаться на основании условий приложения, в котором он будет использоваться.

Таким образом, репликацию целесообразно использовать для адаптивного контейнера, только если количество операций чтения (SELECT) данных намного больше, чем опе-

раций изменения данных (INSERT/UPDATE), поскольку в этом случае будут эффективно использоваться все доступные серверы.

3. КОЛИЧЕСТВО ЗАПРОСОВ

Это второй по важности параметр после объема информации. Возможна ситуация, когда количество данных огромно, а количество запросов к ним невелико. С другой стороны, возможно небольшое количество часто запрашиваемых или часто изменяющихся данных. Для каждого из таких случаев необходимо выбрать свой оптимальный контейнер.

Помимо этого важно понимать количество и типы запросов. Существует три базовых запроса:

1. Поиск (Select). Операция чтения данных из контейнера.

2. Вставка (Insert). Операция вставки данных.

3. Удаление (Remove). Операция удаления данных.

При этом каждый контейнер данных является оптимальным и работает эффективнее других для определенного соотношения запросов. Например, простой список хорошо подходит для большого количества вставок, но плохо для операций поиска и удаления.

В свою очередь, B-дерево не стоит использовать для количества операций вставки примерно равному количеству операций поиска при малом количестве удалений. В остальных случаях этот контейнер хорошо подходит для различных соотношений операций вставки, удаления и поиска.

На основании этих соотношений можно понять, в какой момент необходимо перестраивать контейнер, т. е. адаптировать его под новое соотношение запросов вставки, удаления и поиска. Следовательно, необходимо понять, какое соотношение базовых операций является пограничным, т. е. после преодоления которого необходимо менять контейнер данных.

4. АРХИТЕКТУРА ИНФОРМАЦИОННОЙ СИСТЕМЫ

Еще одним важным фактором при адаптации является архитектура информационной системы (ИС) [22]. Существует несколько типов ИС:

1. Настольные (desktop), или локальные ИС, в которых все компоненты находятся на одном компьютере, а следовательно контейнер данных располагается на этом же компьютере. В этом случае нет никаких дополнительных затрат памяти и процессорного времени на работу оптимального контейнера.

2. Файл-серверные ИС [22]. Данная архитектура показывает себя хорошо в современных условиях при небольших объемах данных, но при увеличении числа компьютеров в сети или росте БД [23] происходит резкое падение производительности. Причина заключается в резком увеличении объема данных передаваемых по сети, так как все данные обрабатываются на компьютере пользователя. Например, когда пользователю требуется несколько записей из таблицы, несколько тысяч строк, файл-сервер передаст всю таблицу на компьютере пользователя и только потом СУБД [24] отберет необходимые записи.

Для адаптации оптимального контейнера – это худший вариант ИС, поскольку при запросе одного элемента из контейнера будет извлекаться весь контейнер. Кроме того, при любом запросе передается большое количество дополнительной информации, что при большом количестве запросов к контейнеру ведет к избыточным затратам процессорного времени и памяти. При этом стоит помнить, что это сервер и клиент – это различные машины и, следовательно, данные передаются по сети, что существенно ограничивает скорость обмена информацией.

3. Клиент-серверные ИС [22]. Сервер обрабатывает запросы клиента и извлекает необходимые клиенту данные из базы данных и возвращает приложению, расположенному на компьютере клиента. Основное преимущество такой архитектуры заключается в том, что количество передаваемых данных значительно меньше, чем в других архитектурах.

Для адаптации контейнера эта архитектура ИС подходит лучше, чем файл-серверная, потому что клиенту передается только та информация, которая им запрашивалась. Но также как и в файл-серверной архитектуре скорость обмена информацией между клиентом и сервером ограничена скоростью передачи по сети.

5. ЗАКЛЮЧЕНИЕ

В данной статье рассмотрена задача построения самоадаптирующихся ассоциативных контейнеров данных и представлен обзор условий, в которых происходит адаптация, включая объем информации, количество запросов и архитектура информационной системы. В результате анализа условий адаптации выявлены основные ограничения, которые необходимо учитывать при реализации самоадаптирующихся ассоциативных контейнеров данных.

СПИСОК ЛИТЕРАТУРЫ

1. *Зобов В. В.* Инструмент для моделирования нагрузки на контейнеры данных / В. В. Зобов, К. Е. Селезнев // Материалы четырнадцатой научно-методической конференции «Информатика: проблемы, методология, технологии», 10-11 февраля 2011 г. – Воронеж, 2014. – Т. 3. – С. 154–161.

2. *Елисеев Д. В.* Аппаратно-программные средства карманных компьютеров / Д. В. Елисеев. – СПб: БХВ-Петербург, 2003. – 368 с.

3. *Jacob B.* Memory Systems: Cache, DRAM, Disk / B. Jacob, D. Wang, N. Spencer. – San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. – 900 p.

4. *Таненбаум Э.* Архитектура компьютера / Э. Таненбаум. – СПб. : Питер, 2016. – 816 с.

5. Архитектуры и топологии многопроцессорных вычислительных систем / А. В. Богданов. [и др.]. – М. : Интернет-университет информационных технологий, 2004. – 176 с.

6. Introduction to Parallel Computing, Second Edition / A. Grama [et al.]. – Boston: Addison Wesley, 2003. – 856 p.

7. Алгоритмы: построение и анализ / Кормен Т. [и др.]. – 2-е изд. – М. : Вильямс, 2005. – 1296 с.

8. *Petersen R.* Linux: The Complete Reference, Sixth Edition / R. Petersen. – NY : McGraw-Hill Osborne Media, 2007. – 830 p.

9. *Solomon D.* Windows Internals, 6th edition / D. Solomon, M. Russinovich. – Redmond, Washington: Microsoft Press, 2014. – 144 p.

10. Oracle RAC & Grid Tuning with Solid-state Disk: Expert Secrets for High Performance Clustered Grid Computing / M. Ault [et al.]. – Kittrell, NC:Rampant Techpress, 2006. – 208p.

11. *Mistry R.* Introducing Microsoft SQL Server 2014 / R. Mistry, St. Misner. – Redmond, Washington: Microsoft Press, 2014. – 144 p.

12. *Вирт Н.* Алгоритмы + структуры данных = программы / Н. Вирт. – М. : Мир, 1985. – 406 с.

13. *Артемов М. А.* Обработка больших объемов данных на основе MapReduce / А. Ш. Исламов, М. А. Артемов, Е. С. Барановский, М. В. Киргинцев // Информатика: проблемы, методология, технологии. Материалы XV международной научно-методической конференции. – Воронеж, 2015. – С. 78–80.

14. *Лацис А. О.* Как построить и использовать суперкомпьютер / А. О. Лацис. – М. : Бестселлер, 2003. – 240 с.

15. *Шилов С. Н.* Реализация инфраструктуры распределенной хеш-таблицы в рамках кластерной системы DNS / С. Н. Шилов, С. Д. Кургалин, А. А. Крыловецкий // Вестник Воронеж. гос. ун-та. Сер. Системный анализ и информационные технологии. – 2012. – № 2. – С. 74–79.

16. *Шилов С. Н.* Двухуровневая схема организации таблиц распределения запросов в кластерной системе DNS / С. Н. Шилов, С. Д. Кургалин, А. А. Крыловецкий // Вестник Воронеж. гос. ун-та. Сер. Системный анализ и информационные технологии. – 2014. – № 1. – С. 90–96.

17. *Ghemawat S.* The Google File System / S. Ghemawat, H. Gobioff, S.-T. Leung // Proceedings of the 19th ACM Symposium on Operating Systems Principles. – Bolton Landing, NY, 2003. – P. 20–43.

18. *White T.* Hadoop: The Definitive Guide, 4th Edition / T. White. – NY : O'Reilly Media, 2015. – 756p.

19. *Кнут Д.* Искусство программирования, том 1. Основные алгоритмы. / Д. Кнут. – 3-е изд. – М.: Вильямс, 2006. – 720 с.

20. Оптимизация и масштабирование Web приложений. – Режим доступа: http://dev.ruhighload.com. – Заглавие с экрана. – (Дата обращения: 14.12.2016).

21. *Шилов С. Н.* Анализ и реализация механизма репликации ресурсных записей в DNS кластере / С. Н. Шилов, С. Д. Кургалин, А. А. Крыловецкий // Информационные технологии. – 2014. – № 6. – С. 38–43.

22. *Советов Б. Я.* Архитектура информационных систем / Б. Я. Советов [и др.]. – М. : Издательский центр «Академия», 2012. – 288 с.

23. *Гарсиа-Молина Г.* Системы баз данных. Полный курс. / Г. Гарсиа-Молина, Д. Д. Ульман, Д. Уидом. – М.: Вильямс, 2004. – 1088 с.

24. *Дейт К. Дж.* Введение в системы баз данных / К. Дж. Дейт. М.: Вильямс, 2006. – 1328 с.

Потапов Данила Романович – аспирант кафедры программного обеспечения и администрирования информационных систем, факультет прикладной математики, информатики и механики, Воронежский государственный университет.
E-mail: potapovd36@gmail.com

Potapov Danila Romanovich – PG student of Chair of Software and Administration of Information Systems, Department of Applied Mathematics, Informatics and Mechanics, Voronezh State University.
E-mail: potapovd36@gmail.com

Артемов Михаил Анатольевич – д-р физ.-мат. наук, профессор, заведующий кафедрой программного обеспечения и администрирования информационных систем, факультет прикладной математики, информатики и механики, Воронежский государственный университет.
E-mail: artemov_m_a@mail.ru

Artemov Mikhail Anatolievich – Doctor of physico-mathematical sciences, Professor, Head of Chair of Software and Administration of Information Systems, Department of Applied Mathematics, Informatics and Mechanics, Voronezh State University.
E-mail: artemov_m_a@mail.ru

Барановский Евгений Сергеевич – канд. физ.-мат. наук, доцент кафедры программного обеспечения и администрирования информационных систем, факультет прикладной математики, информатики и механики, Воронежский государственный университет.
E-mail: esbaranovskii@gmail.com

Baranovskii Evgenii Sergeevich – Candidate of physico-mathematical sciences, Associate professor, Chair of Software and Administration of Information Systems, Department of Applied Mathematics, Informatics and Mechanics, Voronezh State University.
E-mail: esbaranovskii@gmail.com