ПРОГРАММИРОВАНИЕ, 2022, № 5, с. 54–67

БАЗЫ ДАННЫХ, ХРАНИЛИЩА ДАННЫХ

УДК 004.65, 004.658, 519.688

РЕПОЗИТОРИЙ ДАННЫХ В КОНТЕКСТЕ ВЫЧИСЛИТЕЛЬНЫХ ЭКСПЕРИМЕНТОВ: МОДЕЛЬ, АРХИТЕКТУРА, ИНТЕРФЕЙСЫ, ОЦЕНКИ ПРОИЗВОДИТЕЛЬНОСТИ ПРОГРАММНЫХ РЕАЛИЗАЦИЙ

© 2022 г. А. А. Иванков*, Г. А. Мануилов[a],**

[a]*ООО "ДСП Лабс" 194044 Санкт-Петербург, ул. Гельсингфорсская, д. 2, Россия*

**E-mail: a.vnkv1@gmail.com*

***E-mail: george.manuilov@gmail.com*

Поступила в редакцию 04.05.2022 г.
После доработки 11.05.2022 г.
Принята к публикации 18.05.2022 г.

DOI: 10.31857/S0132347422050041

1. ВВЕДЕНИЕ

Работа имеет отношение к такой области исследований, как архитектура и реализации программного обеспечения (ПО), которые пригодны для построения компонентно-ориентированных архитектур конвейерной обработки данных. Заметим, далее речь пойдет об архитектуре и реализациях такой обработки в контексте одного единственного процесса. Научные интересы одного из авторов сосредоточены на решении задач, имеющих отношение к статистике случайных процессов и полей, задач анализа различного рода статистических моделей. Поэтому в качестве контекста, в котором обсуждается предлагаемое нами решение, мы будем использовать формализм и понятия, имеющие отношение к статистике случайных процессов. Реализацию процедуры решения отдельной задачи мы будем называть вычислительным экспериментом (ВЭ). Более конкретное определение этого понятия будет предложено ниже.

2. МОТИВАЦИЯ

На протяжении более чем двух десятилетий в поле нашего зрения попадало различное прикладное ПО (ППО), позволяющее построить ВЭ как конвейерную обработку данных, оперируя программными компонентами посредством публичного API, декларируемого разработчиками этого ППО как стандарт де-факто.

Изо всего множества разнообразных библиотек и фрейворков, с которыми пришлось иметь дело, мы преимущественно выделяли свободно-распространяемое ППО, разработанное на языках программирования C/C++. Упомянем лишь небольшое подмножество в определенном смысле полярных решений: Boost.Accumulators [1], RaftLib [2], R-package [3] и TensorFlow [4].

Выбор субъективен, но он, на наш взгляд, позволяет пояснить те особенности архитектуры, интерфейсов существующих реализаций, которые мы расценивали как недостатки. Предложенное нами собственное решение – попытка устранить эти недостатки.

2.1. Boost.Accumulators

Фреймворк Boost.Accumulators изначально рассматривался как один из прототипов нашего будущего решения. Возможность автоматически построить орграф вычислений искомых оценок – наиболее привлекательный механизм, который реализован в Boost.Accumulators. Однако он не компенсирует следующие недостатки его высокоуровневой архитектуры: 1) построение публичного API на основе статического полиморфизма; 2) очень лаконичная, примитивная, на наш взгляд, система поддержки метаданных, которые ассоциированы как с наборами данных (в контексте Boost.Accumulators это сущность feature – оценка), так и с компонентами орграфа (в контексте Boost.Accumulators это сущность accumulator – аккумулятор). Статический полиморфизм не позволяет спроектировать универсальные, унифицированные публичные интерфейсы как для сущности "компонента", именуемой нами далее по тексту эстиматором, так и для сущности "набор данных", как элемента потока данных, автоматически порождаемого реализацией орграфа вычислений. Система поддержки метаданных, построенная на основе шаблона features⟨⟩ представляется нам весьма ограничительной. Определение синонимов отдельных оценок с помощью шаблонов feature_of⟨⟩, as_feature⟨⟩ – это очень

скромные средства реинтерпретации метаинформации, ассоциированной с конкретной оценкой. Очевидная потребность построения потока данных, содержание которых — многомерные наборы данных, порождает очень сложные задачи как раз в части реинтерпретации компонентами-потребителями отдельных подмножеств таких наборов. Определение функциональных зависимостей между оценками с помощью шаблона depends_on⟨⟩ не позволяет перейти к декларативному способу их определения, не сломав сам механизм автоматического построения орграфа вычислений.

2.2. RaftLib

Код этой библиотеки — обширная система шаблонов. Автоматическое распараллеливание высокоуровневых ETL-операций, которое реализуется кодом самой RaftLib, будет полезно, если только семантика этих операций в задачах обработки данных достаточно тривиальна.

Масштабируемость решений, построенных на основе библиотеки, представляется проблематичной, как только потребуется перейти к обработке многомерных наборов данных, подмножества которых неоднородны с точки зрения их семантики. Статический полиморфизм, положенный в основу архитектуры, существенно осложняет не только переход к проектированию интерфейсов на основе динамического полиморфизма, но и разработку кода компонент согласно PIMPL-принципу.

2.3. R-package

R-package — программное решение, за более чем два десятилетия своего существования ставшее авторитетной альтернативой коммерческим. Оно постоянно пополняется новыми пакетами для статистического анализа данных. С точки зрения конечных пользователей, т.е. с точки зрения функциональных возможностей, мы действительно наблюдаем эволюцию этого пакета. Однако на уровне программных интерфейсов низкого уровня кардинальных изменений нет: в основе интерфейсов кода, написанного на языке C, обширная система макросов, которая является типичным средством его параметризации, в том числе на уровне типов данных. Это одно из самых принципиальных ограничений, которые не позволяют трансформировать существующие интерфейсы так, чтобы переориентировать вектор их развития в направлении компонентно-ориентированных архитектур. Добавим, что и в этом пакете на уровне интерфейсов низкого уровня крайне сложно перейти к организации потока многомерных и неоднородных, с точки зрения семантики, наборов данных. Примечательно, что в документации, которая включена в дистрибутив R-package, мы уже много лет встречаем рассуждения его разработчиков, суть которых: “неплохо было бы попробовать построить код на основе компонентно-ориентированного подхода” (Перевод авт.) [http://cran.org/README, section 4. GOALS.]. Однако конкретных изменений в коде дистрибутива R-package пока не наблюдаем.

2.4. TensorFlow

TensorFlow — фреймворк, предназначенный для построения произвольных графов вычислений, путем их декларативного описания. Граф представляется набором операторов — отдельных программных компонентов, которые оперируют так называемыми тензорами — многомерными массивами данных. Фреймворк TensorFlow предназначен для решения задач машинного обучения, а конкретно — для обучения и инференса (inference) искусственных нейронных сетей (ИНС). Для этих целей TensorFlow успешно комбинирует две концепции: автоматическое дифференцирование и программирование потоков данных. Как и в рассмотренных ранее библиотеках, в TensorFlow передача данных производится напрямую от производителя к потребителю, а также граф вычислений автоматически распараллеливается. К существенным недостаткам TensorFlow в контексте рассматриваемой проблемы можно отнести: ориентированность сугубо на ИНС, сложность высокоуровневого Python API и еще большая сложность аналогичных операций через C++ API, сложность разработки собственных компонентов-операторов. Кроме того, как и все рассмотренные выше программные решения, TensorFlow обезличивает данные при их передаче от оператора к оператору, таким образом не позволяя производить непротиворечивую реинтерпретацию данных в соответствии с их семантикой.

Вышеуказанные особенности (прежде всего низкоуровневых интерфейсов) большинства рассмотренных нами пакетов мы трактуем как недостатки на уровне их архитектуры. Эти фундаментальные расхождения между существующим и проектируемым нами ПО — основные причины, мотивация работы, результаты которой изложены ниже.

3. НЕКОТОРЫЕ АСПЕКТЫ ПАРАДИГМЫ КОМПОНЕНТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

Компонентно-ориентированное проектирование и разработка ПО (Component-Based Development), как философский и технологический подход, было предложено более полувека назад (большинство авторов упоминают в качестве пионерской работы проект конца 60-х годов [5]). В СССР начало развитию компонентно-ориентированного программирования было положено

еще в середине 70-х годов прошлого века. В отечественной литературе эти технологии именуют сборочным программированием, а одни из первых работ были выполнены специалистами научной школы академика В.М. Глушкова [6]. Монография [7], выдержавшая два издания, уже давно стала классическим университетским учебником. Стоит заметить, что один из авторов, Лаврищева Е.М., является сотрудником ИСП РАН, а с самыми последними ее работами читатель может ознакомиться на сайте этого института [8]. Принимая во внимание столь давнюю историю развития этого направления научных исследований, обширную литературу, посвященную компонентно-ориентированному/сборочному программированию, может сложиться впечатление, что большинство проблем решены и эти state-of-art-решения доступны любому разработчику ПО. На самом деле все почти с точностью до наоборот. Круг вопросов, которые приходится решать, намного обширнее подмножества решений, которые уже обрели статус стандартов де-факто и де-юре. Мы попытаемся затронуть лишь отдельные аспекты проектирования архитектуры отдельных компонент, когда весь вычислительный конвейер помещается в контекст одного процесса. Наш интерес к таким архитектурам объясняется достаточно просто – производительность вычислений повышается как минимум на порядок, а, исходя из собственного опыта, чаще всего на два-три порядка (если сравнить с производительностью массовых вычислений на основе механизмов межпроцессного взаимодействия). Как в русскоязычных, так и в англоязычных изданиях, посвященных вопросам компонентно-ориентированного/сборочного программирования, нам не удалось найти работы, результатами которых можно было бы воспользоваться буквально как лекалом: от формального определения семантики интерфейсов до публично доступных исполняемых модулей с реализациями предложенных авторами решений. Но это вовсе не означает, что авторы настоящей статьи претендуют на уникальность своего подхода и ПО. Компонентно-ориентированные архитектуры массовых вычислений в контексте одного-единственного процесса реализованы в огромном количестве программных продуктов (несколько обсудили выше), а их разработчики иногда даже не отдают себе отчет о том, какую же архитектуру они выбрали для проектирования своего ПО, ограничиваясь при подготовке документации описанием пользовательских интерфейсов. Комментарии к API – единственный источник информации для разработчиков. В тех редких случаях, когда авторы ПО все-таки находят силы и время для подготовки статей, в их содержании преобладает изложение вопросов вида “для решения каких задач проектировалось ПО?”, “какие конкретные результаты могут получить специалисты предметной области с помощью этого ПО?”. Это наблюдения в т.ч. и за собственной практикой. Далеко не первая наша реализация компонентно-ориентированной архитектуры в контексте одного процесса (язык разработки – СИ с классами) была описана [9, 10] в предположении, что интерес к ней проявят прежде всего пользователи этого ПО, нейрофизиологи и нейрохирурги.

Таким образом, за все время развития компонентно-ориентированного программирования, наряду с огромным количеством предложенных реализаций, растет и число вопросов, которые остаются открытыми. Мы полагаем, что CBD может играть доминирующую роль в IT-технологиях ближайшего будущего, если наметится существенный прогресс по меньшей мере в следующих трех направлениях: разработка системообразующих критериев и стандартов классификации компонент, стандартизация публичных интерфейсов компонент, разработка стандартов координирующих языков. Наша работа – попытка поделиться своими результатами как рабочим материалом для дискуссий уровня “proof of concept” (обоснование подходов к решению некоторого подмножества вышеперечисленных вопросов).

4. ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ

Отдельная компонента в контексте ВЭ с компоненто-ориентированной архитектурой – программная реализация следующего отображения:

$$\begin{gathered} Estimator(Model, Parameters): \\ INPUT \rightarrow OUTPUT, \end{gathered} \tag{1}$$

где $INPUT$ – множество входных данных, элементы которого в нашей архитектуре – это многомерные наборы данных;

$OUTPUT$ – множество выходных данных этого экземпляра, элементы которого – тоже многомерные наборы данных;

$Parameters$ – параметры этого экземпляра компоненты, предоставленные ей как множество многомерных наборов данных;

$Model$ – экземпляр модели, которая параметризует отображение. В общем случае модели необходимы конкретному экземпляру компоненты как для интерпретации входных данных, $INPUT$, так и для интерпретации параметров алгоритма, $Parameters$.

Формальное описание одной транзакции с участием экземпляра компоненты:

$$\begin{gathered} \{pre(Model, Parameters, INPUT)\} \\ Estimator(Model, Parameters): \\ INPUT \rightarrow OUTPUT\{post(OUTPUT)\}, \end{gathered} \tag{2}$$

где $\{pre(Model, Parameters, INPUT)\}$ – предусловие, предикат, формальное определение условий, которым должны удовлетворять модель, параметры алгоритма и множество входных данных; $\{post(OUTPUT)\}$ – постусловие, предикат, формальное определение условий, которым должно удовлетворять множество выходных данных по окончании транзакции.

Отдельный ВЭ определяется как композиция отображений в смысле (1). Далее, в ходе изложения, мы будем пользоваться и другой моделью ВЭ, определяя последний как орграф, узлы которого – экземпляры конкретных компонент, а ребра – потоки данных.

Сущность "эстиматор" в контексте архитектуры наших ВЭ – синоним понятия "компонента". Мы определяем "эстиматор" как самостоятельную, независимую, параметризуемую реализацию алгоритма вычисления конкретного набора оценок. Свойство "независимость" следует понимать как возможность развернуть ее на рабочей станции независимо от других эстиматоров, включаемых в орграфы ВЭ. Свойство "параметризуемость" тоже следует понимать в широком смысле: конкретный экземпляр эстиматора в ходе ВЭ может быть параметризован как на уровне структур данных (классическая реализация параметризации алгоритмов), так и на уровне кода – один эстиматор может быть параметризован экземпляром другого эстиматора. Совокупность экземпляров эстиматоров/экземпляров программных компонент/модулей, их конфигурационных файлов, определяющих порядок решения этими эстиматорами/модулями конкретной задачи, а также совокупность исходных наборов данных, необходимых для решений этой задачи, будем называть конкретной реализацией ВЭ. Выполнение ВЭ – это, в самом простейшем случае, сессия приложения/процесс в конкретной операционной среде, в которой/котором эстиматорами выполняется обработка исходных наборов данных, промежуточных результатов ВЭ. Порядок обработки определен в конфигурационном файле ВЭ. Выполнение отдельного ВЭ завершается после выполнения вычислений всеми эстиматорами, задействованными для решения задачи. Количество высокоуровневых вычислительных операций (высокоуровневая операция задана отдельной директивой конфигурационного файла эксперимента и выполняется отдельным эстиматором) в ходе выполнения ВЭ может быть как детерминированным, так и недетерминированным. Детерминированное количество высокоуровневых операций соответствует ациклическому орграфу ВЭ (например, эстиматоры указаны в линейном порядке). Недетерминированное количество высокоуровневых операций соответствует орграфу ВЭ с циклами (т.е. в конфигурационном файле ВЭ определяется циклический порядок выполнения этих операций и инварианты цикла).

Конкретное количество эстиматоров в ВЭ и порядок вычислений (последовательно, параллельно или некоторая комбинация этих вариантов) пользователь определяет в конфигурационном файле этого ВЭ. Пример фрагмента конфигурационного файла ВЭ представлен в Приложении 1. Соответственно, в Приложении 2 – директивы конфигурационного файла отдельного экземпляра эстиматора. Таким образом, отдельный ВЭ, точнее, его орграф, определяется в конфигурационном файле этого ВЭ, где помимо топологии орграфа явно или косвенно указаны параметры эстиматоров (узлы орграфа ВЭ). Явное определение параметров эстиматора предполагает, что параметры непосредственно указаны в директивах конфигурационного файла. Неявное определение параметров эстиматора – в директивах конфигурационного файла указан запрос к репозиторию (хранилищу данных ВЭ), после успешного выполнения которого эстиматор получит из репозитория искомые параметры как элементы многомерного набора данных, предоставленного ему репозиторием в режиме доступа "только чтение". Входные данные эстиматора всегда определены в форме запроса к репозиторию (см. директиву REPO_ENTRIES в Приложении 2). Наиболее типичная форма запроса к репозиторию содержит определение суррогатного ключа запрашиваемого набора данных:

$$key : '[' \ PK \ '[' \ list_of_parent_PKs \ ']' \ ']';$$

PK – это уникальный ключ запрашиваемого набора данных в репозитории, *list_of_parent_PKs* – список уникальных ключей наборов данных, содержимое которых было использовано при вычислении элементов набора с ключом *PK*. Оба эти элемента грамматики запросов к репозиторию определяются следующим образом:

$$list_of_parent_PKs$$

$$: (PK | (PK(',' PK)^*));$$

$$PK : UINTEGER;$$

$$UINTEGER : [1-9][0-9]^*;$$

Поэтому еще раз подчеркиваем, в объем понятия "параметры эстиматора" включены в т.ч. и выражения на языке запросов к репозиторию, определяющие содержание запроса этого эстиматора к репозиторию как заявку на доступ к хранимым в репозитории данным.

Сущность "набор данных" (далее НД) в контексте нашей архитектуры – это контейнер, который инкапсулирует как сами данные, так и их метаданные. Собственно данные в наших реализациях сущности НД – это двухмерный массив, подмножества которого имеют различную мощ-

ность и разную семантику. Метаданные НД включают дескриптор эстиматора – источника этих оценок, декларативное определение модели, согласно которой был получен этот НД, дескриптор всей совокупности оценок, хранящихся в этом НД (аналог собирательного обобщения понятия feature в контексте фреймворка Boost.Accumulators), наконец, дескрипторы отдельных подмножеств, далее треков, этого многомерного НД как самостоятельных оценок (аналог понятия feature в контексте фреймворка Boost.Accumulators). Поскольку программные реализации эстиматоров могут иметь различное происхождение (программные продукты различных разработчиков), координирующая роль ПО нашего фреймворка сводится к управлению порядком работы эстиматоров (включая время их жизни) и управлению потоками данных между ними. Предлагаемое нами решение, в части организации потоков данных ВЭ, реализует эти потоки между эстиматорами с обязательным участием посредника – централизованного хранилища НД ВЭ (репозиторий).

Мотивация такой архитектуры достаточно очевидна. Централизованное хранилище данных, порождаемых в ходе ВЭ, как самостоятельная компонента, вполне согласуется с компонентно-ориентированной архитектурой ВЭ. Оно реализует основные функциональные требования, предъявляемые к контейнерам с аналогичной семантикой [11]:

– контроль времени жизни данных;

– контроль доступа к данным;

– реализация механизма транзакций с этими данными;

– эффективное использование двух основных ресурсов вычислительной системы: оперативной памяти и процессорного времени.

В той или иной форме подобные хранилища подразумевают авторы многих программных систем, спроектированных согласно компонентно-ориентированной архитектуре. Например, в [11] отмечают, что в системах распределенной обработки данных, построенных ими на основе компонентно-ориентированной архитектуры, конкретное содержание потоков данных между компонентами – дескрипторы информационных пакетов (информационный пакет, ИП, – это семантический аналог нашего набора данных). Компоненты получают доступ к данным, используя эти дескрипторы, но владельцами ИП сами компоненты не могут быть по целому ряду причин. О реализации репозитория ИП автор [12] упоминает неявно, когда поясняет суть этих причин: делегирование компонентам права владения ИП порождает ряд сложных задач по управлению потоками данных, а их решения будут неэффективными во всех смыслах. Становится ясно, что хранение в НД данных вместе с метаданными обусловлено отчуждением эстиматорами их результатов в репозиторий. С того момента, как НД попадает в репозиторий, хранимые им данные становятся обезличенными с точки зрения их семантики, если только в НД не хранится метаинформация. Поэтому совместное хранение в НД собственно самих оценок вместе с их метаданными – один из наиболее эффективных способов определения семантики данных на стороне их потенциальных потребителей – эстиматоров. Напомним, что транзакция эстиматора, в смысле (2), предваряется вычислением операндов предиката $\{pre(Model, Parameters, INPUT)\}$. В части, касающейся множества входных НД, $INPUT$, эти вычисления построены на операндах – элементах метаданных НД из $INPUT$.

5. МОДЕЛЬ РЕПОЗИТОРИЯ

Принимая во внимание две основные функции репозитория – эффективное хранение НД и предоставление доступа на чтение к хранимым НД, мы можем абстрактно описать работу такого хранилища моделью системы массового обслуживания (СМО). В каждый момент времени состояние такой СМО определяется множеством

$$E = E_{RAM} \cup E_{DISK}, \tag{3}$$

где

$$E_{RAM} = \cup_i DS_i \tag{4}$$

– множество НД, хранимых в этот момент в оперативной памяти (далее кэш репозитория),

$$E_{DISK} = \cup_j DS_j \tag{5}$$

– множество НД, хранимых на внешнем запоминающем устройстве (диске). При этом:

$$E_{RAM} \cap E_{DISK} = \varnothing$$

Эффективное хранение НД в репозитории – это оптимальное разделение множества (3) на два непересекающихся подмножества (4) и (5). Задачу оптимизации в общем виде сформулируем следующим образом. Пусть V_{RAM} – максимальный размер кэша, в байтах. V_{DISK} – объем доступного пространства на ВЗУ, в байтах. Выполняется неравенство:

$$V_{RAM} \ll V_{DISK}$$

Пусть отображение:

$$V : E \to v \in N$$

– размер, в байтах, участка оперативной памяти или дискового пространства, необходимого для хранения НД. Пусть функция времени доступа к НД (к хранимым в нем данным):

$$T : E \to at, \quad at \in R_1,$$

причем время доступа к НД в кэше на несколько порядков (согласно нашим результатам по мень-

шей мере на два порядка) меньше времени доступа к НД такого же размера на диске:

$$\forall DS_i \in E_{RAM}, \quad DS_j \in E_{DISK} :$$
$$V(DS_i) == V(DS_j) \Rightarrow T(i) \ll T(j)$$

Ограничение на размер кэша в общем случае не позволяет разместить в нем все НД ВЭ. Безусловное размещение НД на диске приведет к значительному увеличению времени доступа к данным. В ходе ВЭ эстиматоры, как клиенты СМО, обращаются к репозиторию с заявками двух типов: на размещение в репозитории нового НД (s-тип) и на получение доступа к хранимому в репозитории НД (l-тип). Каждому запросу l-типа соответствует некоторый запрашиваемый НД $DS_l \in E$.

Номер t элемента s_t или l_t в случайной последовательности заявок к репозиторию:

$$R = s_{t=1}^{Ns} \cup l_{t=1}^{Nl} \tag{6}$$

можно рассматривать как момент времени (номер шага), в который репозиторий получил заявку s_t или l_t.

Политика размещения НД – отображение:

$$h : (E_{RAM}, E_{DISK}, v(DS_t)) \rightarrow (E^*_{RAM}, E^*_{DISK}), \tag{7}$$

которое определяет новое состояние репозитория (E^*_{RAM}, E^*_{DISK}) на основе текущего состояния репозитория (E_{RAM}, E_{DISK}) и метрик сохраняемого/запрашиваемого НД (в данном случае – размера этого НД). Последовательность состояний репозитория – это множество пар подмножеств (E_{RAM}, E_{DISK}). При этом последовательность заявок s_t порождает пары (E^*_{RAM}, E^*_{DISK}):

$$((E_{RAM} \subset E^*_{RAM}) \; and \; (E_{DISK} == E^*_{DISK}))$$
$$or$$
$$((E_{RAM} == E^*_{RAM}) \; and \; (E_{DISK} \subset E^*_{DISK}))$$

Последовательность заявок l_t порождает пары:

$$((E^*_{RAM} == (E_{RAM} E^{(d)}_{RAM}) \cup E^{(l)}_{DISK})$$
$$and$$
$$(E_{DISK} \subseteq E^*_{DISK})),$$

где $E^{(d)}_{RAM}$) – подмножество E_{RAM}, состоящее из тех НД, которые вытеснены из кэша на диск, а $E^{(l)}_{DISK}$ – подмножество E_{DISK}, состоящее из тех НД, которые загружены в кэш с диска. Для описания поведения репозитория при получении заявок на чтение, заявок l-типа, определим следующую полную группу событий: кэш-попадание (НС) – запрошенный НД оказался в кэше, кэш-промах (МС) – запрошенный из кэша НД в этот момент в нем отсутствует. Оценка уровня попаданий OHR (object hit ratio) – отношение числа кэш-попаданий к общему числу запросов к кэшу. Пусть

$$OHR = ohr_{t=1}^{Nl} \tag{8}$$

статистика (случайная последовательность) уровня попаданий, которая собрана на некоторых конечных интервалах времени, то есть это статистика, которая получена на основе некоторого подмножества (6), составленного из l-запросов к кэшу. Тогда интегральная оценка:

$$\Sigma(OHR) = \sum_{t=1}^{Nl} (ohr_t) \tag{9}$$

одна из количественных характеристик эффективности хранения НД в репозитории.

Основные потери времени связаны с операциями перемещения/копирования НД между кэшем и диском в ходе обработки заявок l-типа (низкий уровень попаданий в последовательности (8)), и операциями доступа к данным тех НД, которые СМО не смогла разместить в кэше. В последнем случае, когда СМО не смогла разместить в кэше запрошенный НД, причины совершенно объективны – все НД, находящиеся в кэше, все еще используются эстиматорами, следовательно, не могут быть вытеснены на диск, а ограничение на размер кэша не позволяет увеличить его емкость. Другими словами, в подобном случае нет никаких предпосылок для того, чтобы улучшить обслуживание l-заявки. Оптимизация в смысле совокупного времени обслуживания l-заявок возможна путем построения политик размещения, которые минимизируют перемещения/копирование НД между кэшем и диском (и, как следствие, время на выполнение этих операций).

Можно дать альтернативное (7) определение политики размещения. Оно позволит сформулировать задачу оптимизации, оперируя понятиями последовательности заявок к репозиторию, R, и последовательности операций размещения/перемещения НД в репозитории, которые реализуют обработку заявок из R:

$$h : R \rightarrow m,$$

где R – множество последовательностей R заявок к репозиторию, m – множество последовательностей операций перемещения НД в репозитории.

Пусть задан функционал H, отображающий из декартова произведения множества h реализуемых политик размещения и множества R последовательностей заявок во множество интегральных оценок $\Sigma(OHR)$ из (9):

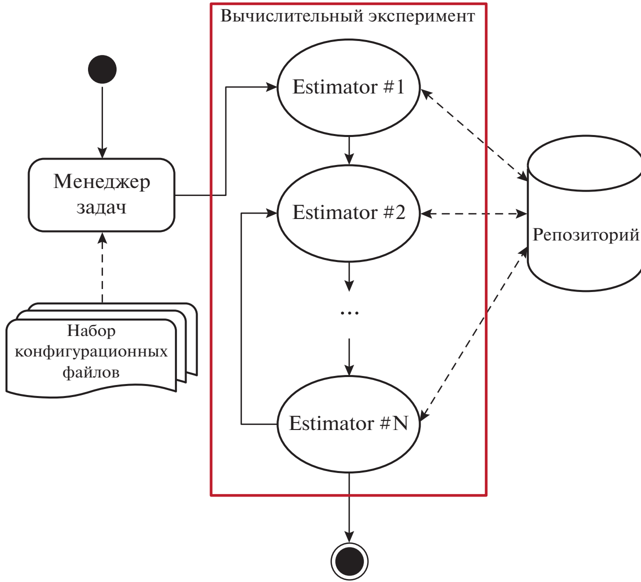

Рис. 1. Архитектура отдельного ВЭ.

$$H : h \times R \to \Sigma(OHR).$$

Тогда для детерминированной последовательности R разработка алгоритмов обслуживания репозиторием заявок в явном или неявном виде может быть сведена к решению следующей задачи максимизации функционала H:

$$h^* = \mathrm{argmax}_h H(h, R),$$

где R – фиксированная последовательность запросов.

В общем случае мы имеем дело со стохастической постановкой задачи оптимизации, поскольку последовательности заявок к репозиторию, R, случайны по своей природе. В таких ситуациях для решения задачи потребуется, во-первых, определить набор инвариантов, которые сохраняются на множестве предполагаемых входных последовательностей R, во-вторых, уточнить определения оптимального решения. Другими словами, необходимо уточнить, в каком смысле мы понимаем оптимальное решение, например, как доставляющее нам максимум математического ожидания оценки $\Sigma(OHR)$ из (9), минимум дисперсии этой оценки, $D[\Sigma(OHR)]$.

5.1. Формальное определение семантики интерфейса репозитория

Определим две аксиомы, которые должны выполняться минимальным публичным API репозитория.

$$\{pre(DS)\} saveData : (E_{RAM}, E_{DISK}, DS) \to$$
$$\to (E^*_{RAM}, E^*_{DISK})\{post((E^*_{RAM}, E^*_{DISK}))\},$$

где $\{pre(DS)\}$ – предикат, формальное определение условия “DS – правильно сформированный экземпляр НД”;

$\{post((E^*_{RAM}, E^*_{DISK}))\}$ – постусловие в виде дизъюнкции: “репозиторий успешно перешел в новое состояние, $((E_{RAM} \subset E^*_{RAM})$ and $(E_{DISK} == E^*_{DISK}))$ or $((E_{RAM} == E^*_{RAM})$ and $(E_{DISK} \subset E^*_{DISK}))$ ” or “репозиторий остался в прежнем состоянии $(E_{RAM} == E^*_{RAM})$ and $(E_{DISK} == E^*_{DISK})$”. Ограничения на объем статьи не позволяют нам уточнить определения операндов каждого из предикатов. Для этого потребуется указать алфавит, который однозначно определяется структурой программных реализаций отдельных сущностей. Операция передачи репозиторию на хранение НД должна быть семантически эквивалентна операции копирования.

$$\{pre(DS_def)\}$$
$$getData : (E_{RAM}, E_{DISK}, DS_def) \to$$
$$\to (E^*_{RAM}, E^*_{DISK}, DS_s c)$$
$$\{post((E^*_{RAM}, E^*_{DISK}, DS_s c))\},$$

где $\{pre(DS)\}$ – предикат, формальное определение условия “DS_def – правильно сформированная строка, идентифицирующая экземпляр НД на основе его метаданных уровня репозитория или уровня самого НД”;

$\{post((E^*_{RAM}, E^*_{DISK}, DS_s c))\}$ – постусловие в виде дизъюнкции: “репозиторий успешно перешел в новое состояние, $((E^*_{RAM} == (E_{RAM} E^{(d)}_{RAM}) \cup E^{(l)}_{DISK})$ and $(E_{DISK} \subseteq E^*_{DISK}))$ или остался в прежнем, $((E_{RAM} == E^*_{RAM})$ and $(E_{DISK} == E^*_{DISK}))$, сформировав shallow copy затребованного НД”, or “репозиторий остался в прежнем состоянии $((E_{RAM} == E^*_{RAM}, E_{DISK} == E^*_{DISK}))$, затребованный НД в репозитории не обнаружен”.

Операция передачи из репозитория хранимого там НД должна быть семантически эквивалентна операции копирования shallow copy.

6. ДВЕ АРХИТЕКТУРЫ РЕПОЗИТОРИЯ ДАННЫХ ВЭ

Поскольку хранилище тоже было спроектировано, следуя концепции компонентно-ориентированного программирования, его архитектуру можно описать, оперируя четырьмя сущностями: справочник репозитория (далее RD), логика кэша (RCL),

кэш репозитория (RC), долговременное хранилище (RPS).

Обработка заявок, поступающих от клиентов – эстиматоров, – это транзакция, совместно осуществляемая экземплярами всех четырех сущностей. Код самого репозитория координирует порядок выполнения транзакций. Другими словами, нашу реализацию хранилища можно рассматривать как транзакционную систему управления многомерными наборами данных. Она (система) размещается, как синглтон, в контекст каждого процесса (см. рис. 1) для того, чтобы контролировать потоки данных между отдельными компонентами ВЭ.

Публичные интерфейсы каждой из четырех компонент составляют всего три операции: runTransaction, commitTransaction, rollbackTransaction. Семантика этих операций в RCL, RD, RC, RPS определяется семантикой конкретной компоненты. Полагаем, она очевидна читателям.

Алгоритм 1: Алгоритм извлечения НД из репозитория

Вход: Уникальный или суррогатный ключ DS в репозитории K

Выход: Shallow copy набора данных DS

```
1 TP := RepoCacheLogic::makeTransactionPlan(K)
2 RepoDirectory::notifyOnTransactionPlan(TP)
3 RepoCache::runTransaction(TP)
4 RepoPersistentStorage::runTransaction(TP)
5 RepoDirectory::commitTransaction(TP)
6 RepoCache::commitTransaction(TP)
7 RepoPersistentStorage::commitTransaction(TP)
8 RepoCacheLogic::commitTransaction(TP)
9 DS := RepoCache::retrieveDS_sc(K)
```

Алгоритм 2: Алгоритм размещения НД в репозитории

Вход: Набор данных DS

```
1 TP := RepoCacheLogic::makeTransactionPlan(DS)
2 RepoDirectory::notifyOnTransactionPlan(TP)
3 RepoCache::runTransaction(TP)
4 RepoPersistentStorage::runTransaction(TP)
5 RepoDirectory::commitTransaction(TP)
6 RepoCache::commitTransaction(TP)
7 RepoPersistentStorage::commitTransaction(TP)
8 RepoCacheLogic::commitTransaction(TP)
```

Алгоритмы представлены в сокращенном варианте, исключены операции восстановления состояния репозитория после сбоя любого из его компонентов.

7. ОСОБЕННОСТИ ПРОГРАММНЫХ РЕАЛИЗАЦИЙ

Нам не удалось найти свободно-распространяемое ПО, которое по своим функциональным возможностям было бы сопоставимо с нашим репозиторием. Чтобы провести сравнительный анализ нашей реализации (далее m-реализация), мы разработали еще и ее альтернативный вариант. Его, по нашему мнению, мог бы спроектировать среднестатистический программист, располагая формальной спецификацией публичного интерфейса (API) репозитория.

Альтернативная, наивная реализация (далее n-реализация) позволяет нам обратить внимание на существенные особенности нашего ПО. Во-первых, в n-реализации кода сущности НД сами данные почти наверняка будут храниться как одномерный массив. Но все метаданные НД: строковые литералы, идентифицирующие эстиматоры, модели, семантика НД и отдельных треков, массивы смещений отдельных треков в экземпляре НД, будут размещаться в нем, не заботясь о возникающей при этом фрагментации оперативной памяти. Наша m-реализация НД гарантирует размещение в памяти всех данных и метаданных НД без фрагментации. Экземпляры m-реализации НД копируемы (copyable) и перемещаемы (moveable) в том же смысле, в котором эти характеристики в стандарте языка C++ указывают для массивов POD-типов [14]. Во-вторых, возможная структура n-реализации кода репозитория почти наверняка, следуя принципу минимализма, – совокупность двух компонент: кэш и долговременное хранилище, где кэш инкапсулирует еще метаданные, код справочника (т.е. RD) и логики хранения (т.е. RCL). Копии экземпляров НД, принимаемых на хранение таким репозиторием, будут размещаться в памяти по наивному сценарию, т.е. не заботясь об устранении фрагментации памяти. Наша m-реализация кода репозитория гарантирует размещение копий НД в RC без фрагментации оперативной памяти. Сама возможность таким образом хранить НД обеспечена вышеуказанными характеристиками m-реализации сущности НД (копируемы и перемещаемы). В-третьих, в n-реализации долговременное хранилище тоже использует механизм файлов, отображаемых в память, но каждый НД сохраняется в отдельном файле. Такой вариант размещения НД на диске основан на наивном предположении о том, что он существенно сокращает время построения представлений на запись и чтение.

По нашим оценкам, только первые две, из этих трех наиболее существенных особенностей m-реализации, снижают совокупный расходуемый репозиторием объем оперативной памяти, по сравнению с n-реализацией, минимум в полтора раза.

8. МЕТОДИЧЕСКИЕ АСПЕКТЫ ОЦЕНИВАНИЯ ПРОИЗВОДИТЕЛЬНОСТИ ПРОГРАММНЫХ РЕШЕНИЙ

В течение всей сессии мы контролируем ту долю адресного пространства процесса, которую используют две основные компоненты в структуре хранилища, RC и RPS, точнее их контейнеры: кэш и представление, используемое для записи НД во внешний файл, проецируемый в память (mmap). Объем используемой ими оперативной памяти ограничен сверху. Во всех четырех компонентах хранилища, в RCL, RD, RC, RPS, имеются собственные локальные справочники. Они представляют собой динамические массивы. По мере накопления метаинформации растет объем используемой ими оперативной памяти, а описать такой процесс можно линейной функцией от количества хранимых в репозитории НД. Эти неизбежные накладные расходы гарантируют логарифмическую временную сложность операции поиска метаинформации в локальных справочниках.

Наибольший интерес, с точки зрения производительности программной реализации наших хранилищ, представляют следующие характеристики:

– средняя длительность транзакции на получение доступа к хранимому в репозитории НД (далее l-транзакция), т.е. среднее время получения shallow copy такого НД;

– средняя длительность транзакции (далее s-транзакция), в ходе которой сохраняем в репозитории новый НД.

Далее мы рассматриваем два типа l-транзакций, поскольку запрашиваемый НД может находиться либо в кэше (RC), либо в долговременном хранилище (RPS). В последнем случае l-транзакция включает в себя еще и операцию загрузки такого НД из RPS в RC.

Два замечания об оценках динамики средней длительности вышеуказанных трех типов транзакций:

1. Мы будем строить такие оценки, интерпретируя накопленный статистический материал как реализации нестационарных с.п. с дискретным аргументом. Для их описания нами были выбраны регрессионные модели, переменная-регрессор которых – порядковый номер транзакции соответствующего типа. В случае s-транзакций их порядковый номер однозначно связан с количеством НД, накопленных к этому моменту в репозитории. В случае l-транзакций такую связь можно трактовать в смысле оценок в среднем.

2. Реализации с.п., которые были получены в результате мониторинга транзакций, содержат достаточно большое количество выбросов. В подобной ситуации классические, неробастные оценки параметров регрессионных моделей в L_2-метрике неустойчивы. Поэтому мы использовали методы робастного регрессионного анализа.

Серии вычислительных экспериментов были спланированы так, чтобы в ходе сессии приложения было обработано не менее 10^4 s-транзакций и не менее 3×10^4 l-транзакций обоих типов. Последовательности таких s- и l-транзакций порождали последовательности s- и l-запросов, в смысле (6). Тип каждого элемента такой последовательности – случайная величина, которую генерировали с помощью биномиального распределения:

$$\forall r_t \in R, \quad P(r_t = s_t) = Bi(1, \theta)$$

Величина параметра θ выбиралась таким образом, чтобы обеспечить вышеуказанное отношение количества s- и l-транзакций. Подобная стохастическая модель потока – попытка построить последовательность случайных переходов между состояниями СМО в смысле (3), когда априорная информация об инвариантах последовательности (6) минимальна. Наборы данных, порождаемые в ходе таких ВЭ, содержали три трека. Это фиксированное число треков на самом деле оценка среднего количества треков в НД, которые порождаются в наших типичных сценариях ВЭ для решения прикладных задач статистического анализа. Размер каждого из треков – это случайная величина из равномерного распределения, а границы носителя меры, $support(track_size)$, заданы как параметры ВЭ. Ниже представлены оценки, полученные в результате четырех серий ВЭ. В каждой серии носитель меры:

$$sup(track_size) := U[1, max_track_size],$$

определяли, выбирая левую его границу, max_track_size, из множества $\{5, 10, 100, 1000, 10000\}$. Другими словами, средний размер НД (в байтах) в каждой серии ВЭ был $\{250, 500, 10^3, 10^4, 10^5\}$ соответственно. Это множество было выбрано опять же исходя из нашего собственного опыта решения конкретных прикладных задач. Статистика была собрана на машинах с процессорами Intel, линейка i5 с тактовой частотой не ниже 3 ГГц, оперативная память – DDR2/DDR3 объемом от 2 до 8 Гб, внешние запоминающие устройства с SATA/PATA – интерфейсами, а операционные системы, выбранные для тестирования, – Windows XP/7, Linux. Одна из версий нашего приложения, позволяющего воспроизвести оценки, предложенные нами в этой работе, находится в свободном доступе [13].

9. ОЦЕНКИ ПРОИЗВОДИТЕЛЬНОСТИ ПРОГРАММНЫХ РЕШЕНИЙ

Вначале представим оценки производительности l-транзакции. Для этого мы выбрали такую

Таблица 1. Оценки границ 95% толерантного интервала для средней длительности транзакции “извлечение shallow copy из репозитория по уникальному ключу НД, когда НД находится в кэше”, мкс

h S	250	500	10^3	10^4	10^5
LRU	7; 13	6; 13	6; 12	7; 16	7; 16
MRU	7; 12	7; 12	7; 13	7; 16	7; 16
GDSF	7; 16	7; 16	7; 16	7; 16	7; 17

Таблица 2. Оценки границ 95% толерантного интервала для средней длительности транзакции “извлечение shallow copy из репозитория по уникальному ключу НД, при условии, что этот НД был помещен в кэш в предыдущей транзакции”, мкс

h S	250	500	10^3	10^4	10^5
LRU	7; 20	7; 20	7; 20	7; 20	7; 20
MRU	7; 12	7; 13	7; 14	7; 17	7; 20
GDSF	10; 12	10; 15	10; 17	10; 25	10; 30

Таблица 3. Оценки границ 95% толерантного интервала для средней длительности транзакции “извлечение shallow copy из репозитория по суррогатному ключу НД, когда НД находится в кэше”, мкс

h S	250	500	10^3	10^4	10^5
LRU	6; 14	6; 12	6; 12	7; 17	6; 17
MRU	6; 9	6; 9	6; 16	6; 16	6; 17
GDSF	7; 16	7; 16	6; 16	7; 19	7; 20

Таблица 4. Оценки границ 95% толерантного интервала для средней длительности транзакции “извлечение shallow copy из репозитория по суррогатному ключу НД, при условии, что этот НД был помещен в кэш в предыдущей транзакции”, мкс

h S	250	500	10^3	10^4	10^5
LRU	6; 15	6; 12	6; 12	7; 13	6; 15
MRU	6; 7	6; 7	6; 7	6; 15	6; 16
GDSF	6; 7	6; 7	6; 7	6; 17	7; 20

Таблица 5. Оценки границ 95% толерантного интервала для средней длительности транзакции “извлечение shallow copy из репозитория по уникальному ключу НД, когда НД находится в кэше”, мкс

h S	250	500	10^3	10^4	10^5
LRU	4; 9	4; 9	4; 10	4; 10	4; 10
MRU	4; 8	4; 8	4; 9	4; 9	4; 9
GDSF	5; 12	5; 12	5; 12	5; 14	5; 16

меру, как 95% толерантные интервалы средней длительности транзакции соответствующего типа, и ограничились тремя хорошо известными реализациями алгоритмов политики размещения НД. В настоящее время наша m-реализация хранилища параметризуема более чем десятком различных алгоритмов политики размещения, большая часть из которых разработана нами. Но оценки производительности транзакций во всех случаях сопоставимы по порядку величин. В приведенных таблицах средний размер НД, в байтах, обозначен как S, а политика размещения – как h.

Нетрудно видеть, что средняя длительность l-транзакций незначительно увеличивается с ростом среднего размера НД. Более высокие оценки в случае GDSF-политики объясняются тем, что в ее алгоритме обработки l-транзакций выполняется еще и сбор статистики, регистрируется количество запросов на доступ к НД. На это дополнительно расходуется, по нашим оценкам, от 2 до 6 мкс.

Для сравнения приведем аналогичные оценки, полученные для альтернативной, наивной архитектуры хранилища и ее n-реализации.

В случае n-реализации границы 95% толерантных интервалов несколько ниже по сравнению с нашей m-реализацией. Это объясняется тем, что во втором случае в m-реализации хранилища дополнительное время (т.е. помимо копирования shared_ptr) расходуется на то, чтобы снабдить shallow-copy оригинального экземпляра НД, хранимого в RC, экземпляром сущности ControlBlock. Этот тип данных спроектирован нами для контроля операций доступа shallow-copy к данным, хранящимся в оригинальном НД.

Мы трактуем приведенные выше оценки как обоснование следующего вывода: средняя длительность l-транзакций определяется константой, которая незначительно растет по мере увеличения размера НД (последнее – в логарифмической шкале).

Далее речь пойдет об оценках для l-транзакций, когда запрашиваемый НД находится в долговременном хранилище (RPS). В таблице 9 они приведены для m-реализации хранилища, в таблице 10 – для n-реализации.

В серии экспериментов, которые проводились для сбора этой статистики, в ряде случаев наблюдаем появление линейной зависимости средней длительности такой транзакции от количества НД в репозитории. Этот рост мы объясняем деградацией производительности системного механизма файлов, отображаемых в память, когда по мере увеличения объема файла увеличиваются смещения, по которым необходимо построить представления.

Таблица 6. Оценки границ 95% толерантного интервала для средней длительности транзакции “извлечение shallow copy из репозитория по уникальному ключу НД, при условии, что этот НД был помещен в кэш в предыдущей транзакции”, мкс

h *S*	250	500	10^3	10^4	10^5
LRU	4; 5	4; 5	4; 11	4; 11	4; 11
MRU	4; 5	4; 5	4; 12	4; 12	4; 12
GDSF	7; 8	6; 12	7; 19	7; 19	7; 20

Таблица 7. Оценки границ 95% толерантного интервала для средней длительности транзакции “извлечение shallow copy из репозитория по суррогатному ключу НД, когда НД находится в кэше”, мкс

h *S*	250	500	10^3	10^4	10^5
LRU	16; 35	15; 20	16; 45	16; 45	30; 70
MRU	15; 20	15; 20	15; 45	16; 50	20; 70
GDSF	16; 38	17; 25	17; 50	17; 50	25; 70

Таблица 8. Оценки границ 95% толерантного интервала для средней длительности транзакции “извлечение shallow copy из репозитория по суррогатному ключу НД, при условии, что этот НД был помещен в кэш в предыдущей транзакции”, мкс

h *S*	250	500	10^3	10^4	10^5
LRU	16; 30	16; 20	16; 50	16; 50	20; 60
MRU	15; 22	16; 20	16; 50	16; 50	20; 60
GDSF	17; 30	17; 20	17; 50	18; 50	20; 70

Как видим, в случае наивной реализации хранилища длительность l-транзакций этого типа значительно растет по мере увеличения размеров НД.

Оценки динамики s-транзакций демонстрируют линейные зависимости от количества хранимых в репозитории НД.

Оценка минимального времени s-транзакций, которая в большинстве серий экспериментов получена на участках, где количество хранимых в RPS НД не превышает 10^2, находится в пределах 200–300 мкс. Коэффициент, определяющий линейную зависимость среднего времени s-транзакций от числа хранимых НД, имеет порядок 2×10^{-2}–$3 \times \times 10^{-2}$. Это означает, что после накопления в файле RPS более 10^3 НД будет наблюдаться значимое (на 20–30 мкс) увеличение средней длительности s-транзакций.

Чтобы понять природу такой связи, необходимо принять во внимание семантику s-транзакции, реализуемой в коде RPS. Обработка s-транзакции в RPS начинается операцией поиска метаданных полученного НД в локальном (уровня RPS) справочнике. Если такой НД зарегистрирован в справочнике RPS, то s-транзакция на этом и заканчивается (т.к. НД уже хранится в RPS). Временная сложность такого поиска – логарифмическая, длительность этой операции в наших сериях ВЭ вырастает от нескольких микросекунд до двух-трех десятков микросекунд. Основные затраты, по времени, приходятся на непосредственно копирование (memcpy) НД во внешний файл, проецируемый в память. Такое копирование периодически предваряется операцией динамического изменения (увеличения) размера внешнего файла, а также последовательностью (атомарной транзакцией уровня RPS) из двух операций: закрытие текущего представления на запись и создание нового представления (по новому смещению). Динамическое расширение размера внешнего файла – это еще один механизм экономии вычислительных ресурсов. Пользователь, при желании, может указать в конфигурационных файлах ВЭ начальный размер файла RPS в несколько десятков гигабайтов, тем самым вовсе исключив операции динамического расширения этого файла. Перестройка представления на запись выполняется достаточно часто. Этот объект позволяет нам сократить до минимума время копирования НД в файл, но за счет потери той части адресного пространства, в которую представление отображается. Становится ясно, в структуру оценки средней длительности s-транзакции входят по меньшей мере три слагаемых: длительность самой операции копирования, длительность операции динамического увеличения файла RPS (редкое событие), длительность операций по перестройке представления для записи в файл (повторяются периодически). Согласно накопленной нами статистики, именно последнее слагаемое определяет линейную зависимость s-транзакций от количества хранимых в репозитории НД, точнее, от совокупного размера НД, хранимых в этом момент во внешнем файле.

Более того, перестройка представлений на запись (согласно статистике, полученной в результате мониторинга s-транзакций), довольно часто имеет характер переходного процесса в этой СМО. Представить адекватное статистическое описание таких участков без тщательного анализа кода, реализующего механизм файлов, отображаемых в память (уровень системных библиотек), не представляется возможным. Но такой анализ не входит в настоящее время в круг наших первоочередных задач.

10. ЗАКЛЮЧЕНИЕ

Предвосхищая возможные замечания, касающиеся приведенных в статье оценок производительности наших программных реализаций, мы согласимся, что они не покрывают варианты конфи-

Таблица 9. Оценки границ 95% толерантного интервала для средней длительности транзакции “извлечение shallow copy из репозитория по уникальному ключу НД, когда НД загружается в кэш в этой же транзакции”, мкс

h *S*	250	500	10^3	10^4	10^5
LRU	170; 400	110; 385	120; 500	125; 10^3	150; 1.2×10^3
MRU	180; 325	100; 380	100; 400	110; 950	130; 1.1×10^3
GDSF	175; 370	165; 310	90; 400	100; 2750	120; 2.3×10^3

Таблица 10. Оценки границ 95% толерантного интервала для средней длительности транзакции “извлечение shallow copy из репозитория по уникальному ключу НД, когда НД загружается в кэш в этой же транзакции”, мкс

h *S*	250	500	10^3	10^4	10^5
LRU	200; 1.6×10^3	180; 1.7×10^3	320; 6.5×10^4	500; 8.8×10^4	1460; 9×10^4
MRU	190; 10^3	180; 1.5×10^3	220; 4.9×10^4	500; 8×10^4	1490; 7×10^4
GDSF	200; 1.35×10^3	185; 1.6×10^3	300; 7.1×10^4	500; 8.6×10^4	1500; 9.2×10^4

гураций аппаратного обеспечения (АО), на котором возможно выполнение ВЭ. Однако заметим, что, во-первых, методика построения множества таких конфигураций определяется множеством доступного АО или тем его конкретным подмножеством, которое предполагают использовать для выполнения ВЭ. Во-вторых, мы приводим оценки производительности не в качестве конкретных ориентиров для конечного пользователя программной реализации хранилища, а как экспериментальное обоснование выводов, касающихся временной сложности транзакций всех типов. Если средняя длительность l-транзакции определяется константой, то на машине с другим аппаратным обеспечением мы можем иметь порядок, отличный от указанного в наших таблицах (даже если в последнем случае мы использовали те же параметры конфигурации экземпляра хранилища и выполняли ВЭ согласно тем же сценариям). Однако эта оценка средней длительности l-транзакции останется константой. Установив на машине SSD-накопители, пользователь может уменьшить порядок частного двух типов l-транзакций, но

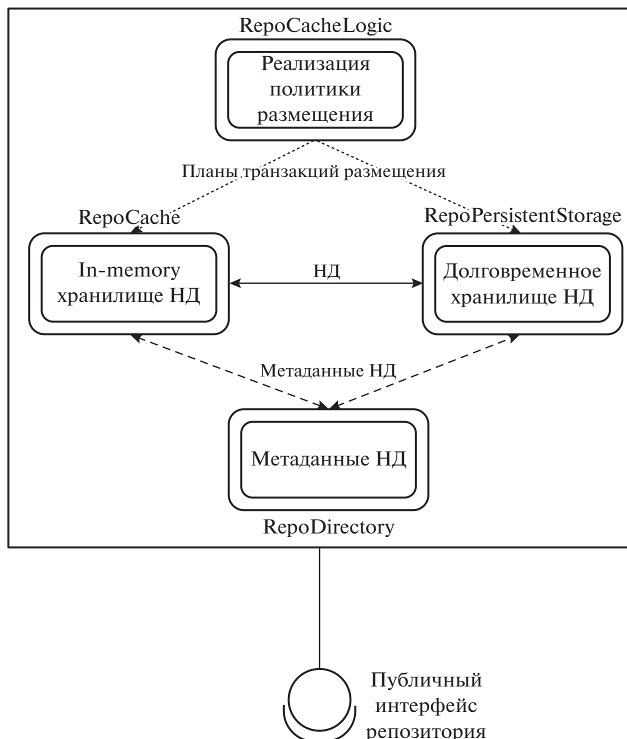

Рис. 2. Архитектура нашего репозитория.

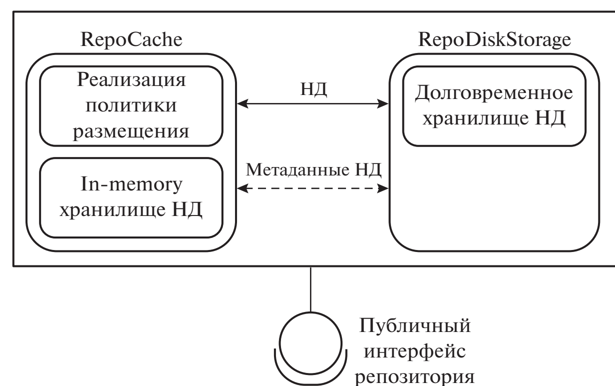

Рис. 3. Архитектура наивной реализации репозитория.

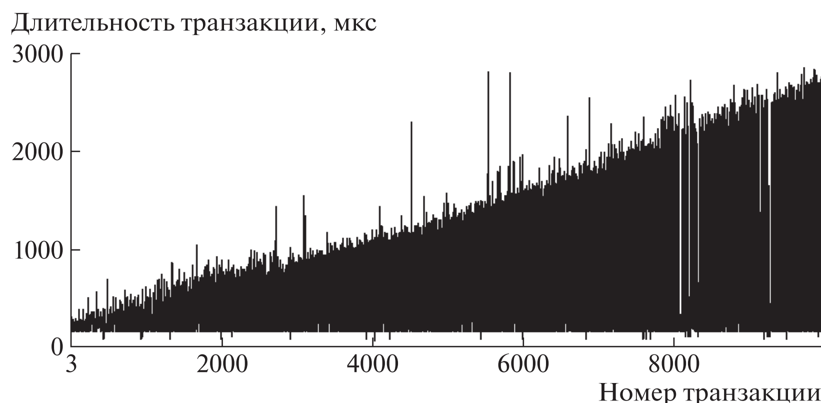

Рис. 4. Динамика длительности s-транзакций. Типичные оценки, полученные нами в одном из ВЭ.

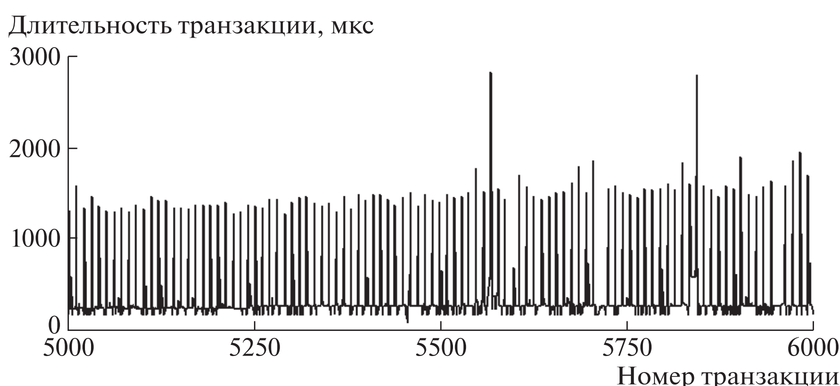

Рис. 5. Один из участков реализации, представленной на рис. 4, как иллюстрация структуры регистрируемого с.п.

сам характер этих зависимостей от количества хранимых в репозитории НД останется прежним. Прикладную значимость нашего программного решения нисколько не умаляет различие на два порядка средней длительности двух типов l-транзакций: в одном случае необходима предварительная загрузка затребованного НД из RPS в RC, а во втором случае в ней нет необходимости, т.к. затребованный НД уже размещен в RC. Напомним, что потери времени на загрузку НД из RPS в RC несравнимы с потерями времени на доступ к элементам этого НД, если его не удается разместить в кэше хранилища и доступ к данным осуществляется последовательными операциями чтения с внешнего запоминающего устройства. В заключение остается надеяться, что результаты нашей работы представляют интерес для достаточно широкого круга IT-специалистов, чья профессиональная деятельность включает в себя решение подобных задач, а неизбежная критика наших результатов будет конструктивной.

11. БЛАГОДАРНОСТИ

Один из авторов хотел бы выразить благодарность своим ученикам и коллегам: Андрею Олеговичу Гаврилову и Артёму Сергеевичу Тетюхину, которые в разное время и в разной мере принимали участие в проектах, имеющих отношение к этой работе.

Работа выполнялась нами как инициативная без какой-либо поддержки (финансовой или в иной форме) от каких-либо юридических или физических лиц. Для реализации изложенных в этой статье алгоритмов нами использовалось только свободно распространяемое программное обеспечение с открытым исходным кодом (CMake, CodeBlocks, gcc).

Приложение 1.
Пример фрагмента конфигурационного файла ВЭ

интервальная оценка параметра пуассоновской модели

STEP_2 = task_cfg/POISSONLAMBDA/taskStep2.cfg

Приложение 2.
Содержимое конфигурационного файла
task_cfg/POISSONLAMBDA/taskStep2.cfg.

# определение входных данных эстиматора на языке запросов к репозиторию
REPO_ENTRIES = [1 [0]]; [2 [0]]

# определение типа эстиматора
ESTIMATOR_TYPE = INTERVAL_ESTIMATOR

# строковый литерал, определяющий конкретную программную реализацию эстиматора
ESTIMATOR_NAME = IntervalPoissonLambda

# имя файла, содержащего параметры эстиматора
ESTIMATOR_PARAMS_SRC = task_cfg/POISSONLAMBDA/IntervalPoissonLambda.cfg

# декларативное определение класса модели, которой параметризуем эстиматор
MODEL_INTERFACE_TYPE =ProbabilisticModelCompositionInterface

# декларативное определение модели, которой параметризуем эстиматор
MODEL_NAME = POISSON_MODEL

СПИСОК ЛИТЕРАТУРЫ

1. Boost. http://www.boost.org
2. RaftLib. http://raftlib.io
3. R-package. http://www.cran.org
4. *Dean J. et al.* TensorFlow: A system for large-scale machine learning. – Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation. 2016. P. 265–283.
5. *McIlroy M.D.* Mass-produced software components. Eds. Naur, Randall. Proceedings, NATO Conference on Software. 1969. P. 88–98.
6. *Лаврищева Е.М.* Развитие отечественной технологии программирования. Кибернетика и системный анализ. 2014. Т. 50. № 3. 16 с.
7. *Лаврищева Е.М., Грищенко В.Н.* Сборочное программирование. Основы индустрии программных продуктов: 2-изд. Дополненное и переработанное. Киев: Наук. думка, 2009. 372 с.
8. https://www.ispras.ru/lavrishcheva/
9. *Иванков А.А.* Программный комплекс для изучения механизмов авторегуляции транскраниального кровообращения в режиме реального времени. Научно-технические ведомости СПбГПУ. Физико-математические науки. 2014. № 3(201). С. 92–109.
10. *Ivankov A.A.* Software platform for real time investigation of cerebral hemodynamics. Humanities and Science University Journal. 2014. V. 10. P. 37–49.
11. *Szyperski Clemens, Gruntz Dominik, Murer Stephan* Component Software Beyond Object-Oriented Programming. Second Edition. Addison-Wesley, 2002. P. 589,
12. *Morrison J. Paul* Flow-Based Programming: A New Approach To Application Development. Second Edition, J.P. Morrison Enterprises, Ltd, 2011
13. https://github.com/flower-flavour/embedded-repo
14. https://en.cppreference.com/w/cpp/language/move_-constructor, абзац “Trivial move constructor”