Авторский перевод статьи Иванкова А.А., Мануилова Г.А. "Репозиторий данных в контексте вычислительных экспериментов: модель, архитектура, интерфейсы, оценки производительности программных реализаций" // Программирование, 2022, No 5, с. 54–67.

Abstract

A data repository design is discussed. The data storage is supposed to be a component of computation pipeline composed of separate computation units. All the components run in the single process context. We discuss repository models, architectures and three implementations. Performance estimates are provided for our implementations.

DATA REPOSITORY IN FRAMEWORK FOR COMPUTATION PIPELINING: MODEL, ARCHITECTURE, THE IMPLEMENTATION PERFORMANCE ESTIMATES

Alexei A. Ivankov*, Georgiy A. Manuilov**,
E-mail: a.vnkv1@gmail.com, george.manuilov@gmail.com

September 19, 2022

1. INTRODUCTION

Our work is related to Component-Based Development (CBD) of software for computation pipelining. The software architecture is assumed to constrain its possible implementations being executable in a single process context At application level the component semantics is explained according to academic interests of authors. Over many years one of authors deals with methods of stochastic analysis. In pipeline a single computation unit is deemed an estimator or standalone software component to evaluate statistics, estimates, test statistical hypothesis, etc. Hereafter we use the notion "Computational Experiment" (CE) for any pipeline instance as such the instance may be an implementation of computer simulations to study a stochastic process, estimation algorithms, etc. Strict definition for CE will be given in what follows.

2. MOTIVATION

Over two decades we have being working with miscellaneous software for computation pipeling. Such implementations may be composed into pipeline in a way that complementary components interact via their interfaces (API & ABI). The following review of a few frameworks, Boost.Accumulators [1], RaftLib [2], R-package [3] and TensorFlow [4]. developed in C/C++, is provided merely to make accent on design flaws we tried to avoid. It's our subjective selection and not the only one possible.

2.1. *Boost.Accumulators*

The Boost.Accumulators (BA) was treated as one of a prototypes for our own framework. Automatic composition of components (so-called accumulators) in a computation pipeline is the most appealing trait of the BA. On the other hand its architecture far from perfect one. We name just a few flaws in BA design: 1) templates buried any possibility to develop concise API, i.e. based on dynamic polymorphism; 2) oversimplified support for metadata on features (estimates in terms of stochastic analysis) and computation units – accumulators. Static polymorphism is not the right base (e.g. see architecture of any-container) to build unified and minimal (in algebraic sense) public interfaces of components as well as interfaces of data containers which the data flow comprises of. Template features<> is the only one to develop consistent metadata support. In our experience the multidimensional datasets are the most appropriate elements of data flow between producers and consumers in pipeline. The BA's feature_of<>, as_feature<> are far from comprehensive support for data semantics reinterpretation which is required in dataflow architectures. Template depends_on<> is very restrictive and any declarative approach / language to define functional dependencies between estimates (features) will break the current automatic mechanism of BA's pipeline composition.

2.2. *RaftLib*

The library is a vast collection of templates. Out-of-box parallelization of ETL-operations is very handy as long as the operation semantics is trivial one. Any solutions based on the library code are scalable for homogeneous dataflow only. There is no chance to develop scalable pipeline once components deal with multidimensional, heterogeneous datasets. Static polymorphism in its turn does not leave any opportunity to develop component's interfaces according to PIMPL-principles, i.e. based on dynamic polymorphism.

2.3. *R-package*

The R is a great project developed for stochastic analysis. Over more than two decades one observes the non-stopping growth of the software at high-level. The low-level interfaces have been reimplemented twice to our best knowledge. In spite of such tremendous efforts the core (C code) still keeps extensive subset of preprocessor macroses. They are the typical tool to parametrize C-code. Unfortunately macro-based development does not leave many opportunities for true CBD which is obvious for R-developers (many years we read their confession in http://cran.org/README, section 4. GOALS). Meantime we are not aware of any changes in R-code/ advances towards CBD-architectures.

2.4. *TensorFlow*

The framework provides the programming environment for developing pipelines as computation graphs (actually digraphs). Graphs are defined in a declarative language. The computation units, graph vertices are operators in TensorFlow terms. The dataflow comprises multidimensional datasets (so-called tensors). The most typical applications are developed for machine learning and related problems. The implementation combines automatic differentiation of goal functions along with effective management of dataflows. There is out-of-box computation parallelization mechanism. The main design flaws: 1) component (operator) development is rather cumbersome with TensorFlow's PyAPI and gets much more painstaking with C++ API; 2) tensors are abstract multidimensional arrays, there are not any metadata and no chances to provide/get data semantics at operator-producer/operator-consumer sides.

The aforementioned design drawbacks manifest themselves at API level and leave us no chance to develop software according to a few mandatory principles: 1) unified, concise interfaces; 2) PIMPL/opaque implementation; 2) declarative definition of computational digraph. Instead of coping with flaws and drawbacks of numerous frameworks we developed our own one. This paper presents some results of our research.

3. SOME ASPECTS OF COMPONENT-BASED DEVELOPMENT

To our best knowledge the first paper on CBD appeared more then forty years ago [5]. In USSR the development of component-based technologies (so-called assembling programming) was initiated by academician V.M.Glushkov and his scholars [6]. The two editions of monograph [7] were included in syllabuses of university courses long time ago. Lavrishcheva, E.M. is one of the monograph authors. Meantime she works at IPS RAN and her recent papers on the matter one may find at the institute website [8]. Whereas the CBD technologies have been studied over half a century the CBD standards are still far from state-of-art. The reason is simple: the research area encompasses many aspects. We consider the only one: the computation pipelining implementations developed as composition/superposition having its components are designed to run in single process context (hereafter single process pipelining). Such pipelines outperform ones based on interprocess communications more than 10 times. In out experience the single process pipelining outperforms 100-1000 times the ones designed to run components in separate processes. There are a few bottlenecks of single process pipelining. In this paper we tackle the only one – data storage in pipeline context. To our best knowledge similar architectures of data storages were implemented in variety of ways: numerous standalone solutions, a part of some software, etc. Unfortunately their authors confined

the software documentation to user interface manuals. In those rare cases the papers have been prepared they were intended preferably for end-user and focused on end-user demands and not software developer ones. We blame our practice as well. Our hardly the first implementation of CBD-pipeline (developed in C with classes) was described [9,10] merely to familiarize neurophysiologists and neurosurgeons with our software facilities. The variety of problems remains open so far. We believe CBD will dominate in software development technologies on condition the advances in the following three areas: component classification and the classification criteria standardization, standardization of public interfaces of components, pipeline COORDINATING language standardization. The paper is our attempt to share the thoughts and results on data storage architectures for single process pipelining. One may take it as RFC at proof-of-concepts level.

4. MAIN DEFINITIONS

A pipeline component is assumed to be the implementation of the following mapping:

$$Estimator(Model, Parameters) :$$
$$INPUT \rightarrow OUTPUT, \quad (1)$$

where $INPUT$ – set of input data, comprised datasets (see the definition below);
$OUTPUT$ – set of output data, comprised datasets;
$Parameters$ – the parameters of the algorithm implemented as Estimator, comprises datasets as well;
$Model$ – the instance of model (definition see below) to parametrize the instance of Estimator. In general case an instance of model is required to interpret $INPUT$ or $Parameters$ data. One may treat Estimator as acting agent. Its atomic act is a transaction which consumes and processes $INPUT$ to produce $OUTPUT$.

Formal definition of such transaction:

$$\{pre(Model, Parameters, INPUT)\}$$
$$Estimator(Model, Parameters) :$$
$$INPUT \rightarrow OUTPUT\{post(OUTPUT)\}, \quad (2)$$

where $\{pre(Model, Parameters, INPUT)\}$ - precondition, predicate or a formal definition of conditions that must be met by the $Model$, $Parameters$ and $INPUT$;
$\{post(OUTPUT)\}$ - postcondition, predicate or a formal definition of conditions that must be met by the transaction $OUTPUT$.

A "Computational Experiment" (CE) is a composition of mappings in sense that defined by (1). In general case it's a composition of superpositions.

Alongside that CE definition we will use another one: a directed graph comprises Estimators-vertices and dataflows-edges. It's clear the Estimator entity is treated as the synonym for component entity. We define Estimator as a standalone, separate implementation (parametrized in general case) of an algorithm which computes some set of estimates. The property "standalone" means any deployments of Estimators on a workstation are independent. The property "parametrized" we define as follows: an Estimator instance can be parametrized in common sense (i.e. with data structures to define algorithm parameters) as well as in general case (i.e. with model instances to interpret input data, parameters or with instance(s) of another Estimator(s) to implement superposition). CE instance is a collection of Estimator instances, their config-files (descriptions of the jobs the Estimator instances are supposed to accomplish, to put it simple the order of computations or CE-digraph verticis and edges) and set of input data. That collection is prepared to solve a problem (to carry on estimation procedures and produce estimates required). CE execution is a running process/application instance as the context to run Estimators of CE instance according to the CE config-files in order to accomplish CE task/solve a problem. CE ends once all the CE Estimators have accomplished their evaluation. The CE output comprises the OUTPUT of the Estimators accomplished their parts of CE task. The transaction of an Estimator (in sense that defined (2)) is a high-level operations specified in corresponding config-file. The total number of high-level operations may be unknown apriori in case of nondeterministic CE, i.e. CE digraph with cycles. Deterministic CE (i.e. CE acyclic digraph) ends after apriori specified number of high-level operations accomplished. Nondeterministic CE is defined by config-file which contains LOOP-directive(s) and its (their) invariant(s) / predicate(s) may include some OUTPUT as operand(s), Deterministic CE is defined by config-file without LOOP-directive(s) and total number of operations is equal the number of Estimators which are composed as pipeline. It's clear the CE instance (its digraph) is defined by CE config-file and the ones of CE Estimators. An example of single directive from CE config-file see in Addendum 1. The Addendum 2 shows an Estimator instance config-file. There are two ways to define an Estimator parameters into its config-file: explicit definition according to the grammar declared in the public API of the Estimator; implicit definition by index of the dataset in CE data repository. In the former case the Estimator code parses the directives and extracts its parameters from the config-file. In the latter case the Estimator code requests its parameters from CE data repository. Once the query has been successfully processed, the CE data repository provides access (read-only) to the dataset requested. Estimator input data are specified in its config-file as query to CE repository, as a set of dataset indexes in CE data repository (see directive REPO_ENTRIES in Addendum 2). The typical query contains a surrogate key of the dataset requested:

$$key: \ '[' \ PK \ '[' \ list\_of\_parent\_PKs \ ']' \ ']';$$

PK - unique, primary key, index of the required dataset in CE repository, $list\_of\_parent\_PKs$ - a list of unique, primary key, index of the datasets (in CE repository) that have been input data to produce the required one (the dataset with index PK). The grammar for both PK and $list\_of\_parent\_PKs$:

$$list\_of\_parent\_PKs$$
$$: (PK|(PK(',' PK)*));$$
$$PK : UINTEGER;$$
$$UINTEGER : [1-9][0-9]*;$$

Recall the implicit definition of Estimator parameters is prepared according to the same grammar. The semantics of such queries is the request for read-only access to dataset(s) stored in CE repository.

The entity "dataset" (hereafter DS) plays the role of the container to keep data in CE context and carry out their exchange across CE Estimators. The container encapsulates data along with their metadata. DS keeps data as jagged 2D array (i.e. an array of arrays such that member arrays can be of different sizes). The semantics of 1D members are different in general case. DS metadata comprises: Estimator token to identify the Estimator produced the DS, declarative definition of the model which was in use to produce the DS, DS token to identify that DS (an Estimator may produce more than one DS), tokens for all the 1D-arrays (hereafter DS tracks) to identify the estimates the tracks keep (one may treat a track as feature in Boost.Accumulators context). Our framework architecture implies the binary files (DSO) of any Estimator may be deployed on workstation in run-time (framework runs such binaries once dlopen() accomplished). There is no need to build Estimator DSO along with framework code. The COORDINATING role of the framework code in run-time is confined to 1) control Estimator instance life-time; 2) control dataflows. All the dataflows in CE context are implemented as producer-consumer interaction and CE data repository is mandatory participant (i.e. producer or consumer).

The reasons behind such architecture are obvious. Repository-singleton to keep CE data is a standalone component and meets the CBD-based architecture of the CE instance. The repository implements the main functional requirements for containers with the same semantics [11]:
- DS life-time control;
- data access control;
- mechanism of DS transaction;
- efficient usage of two main resources: RAM and CPU-time.

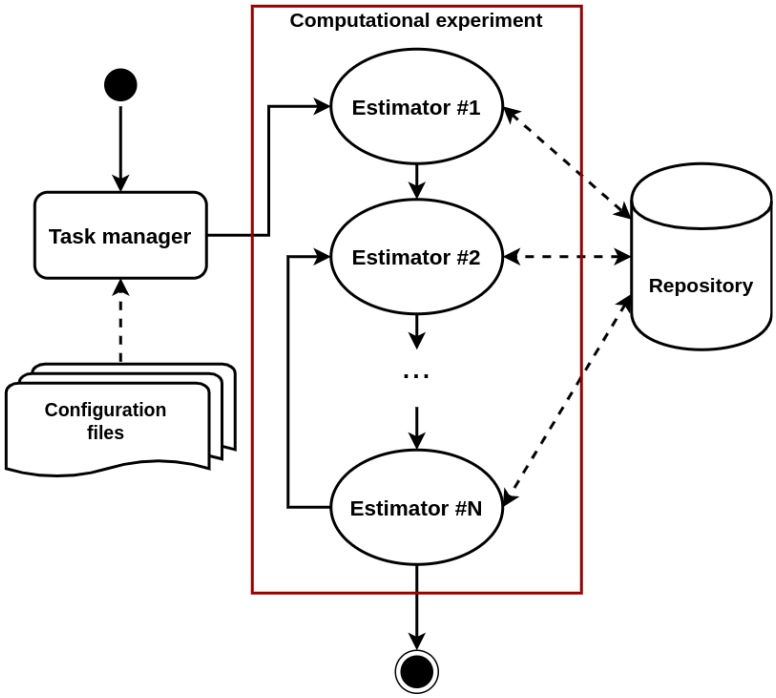

Figure 1: A sample CE architecture.

The similar data storages were mentioned by numerous authors of software which met CBD- principles. The distributed data processing in [11] was an implementation of dataflow system where dataflows comprised descriptors ("handles" in [11]) of "information packets" ("IP"). "IP" semantics is the same one of our DS. Components get access to "IP" data via its "IP handle" but can not possess the latter. Author [12] mentioned about repository implementation while reasoning on the withdraw the IP possession right from components: the decision to grant a component the IP possession right gives rise to severe problems and all possible solutions of the problems are far from efficient in any sense. An Estimator may dispose its OUTPUT DSs once their copies have been provided to CE repository. An Estimator-consumer is not aware of its Estimator-producer which may not exist at the moment of the consumption. That's why the DS metadata are the only one source on DS data semantics for Estimator-consumer. Recall the transaction in sense defined by (2) can be processed by Estimator-consumer if and only if the predicate $\{pre(Model, Parameters, INPUT)\}$ yields true. The predicate operands considering $INPUT$, are INPUT's DSs metadata tokens.

5. REPOSITORY MODEL

Taking into account both the main roles of repository in CE architecture: efficient DS storage and DS access controller, we choose the repository model

as queuing system (QS) one. At any given moment the QS state is defined by following set

$$E = E_{RAM} \cup E_{DISK} \quad (3)$$

, where

$$E_{RAM} = \cup_i DS_i \quad (4)$$

- the set of DSs which are at RAM (hereafter repository cache) at that moment,

$$E_{DISK} = \cup_j DS_j \quad (5)$$

- the set of DSs which have been saved on hard disk. The following condition holds at any moment of time

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

Efficient DS storage must guarantee an optimal split of set (3) into two mutually exclusive subsets (4) and (5) at any moment. Optimization problem is posed as follows. Let V_{RAM} – maximum size of cache, in bytes. V_{DISK} - available room on hard disk, in bytes. The inequity holds for sure:

$$V_{RAM} << V_{DISK}$$

Let mapping:

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

- the size, in bytes, of continuous region in RAM/on hard disk to keep a DS which belongs set (3). Let DS access time function (the time it takes to get access to DS's data or metadata):

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

Time to get access to DS in cache less (according to our estimates at least 100 time less) than the time to get access to a DS (of the same size) on disk:

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

Cache size limit does not allow to store all DSs in the cache. DS access time will be enormous if we save all the DSs on disk. The running CE Estimators are SQ-clients which request for CE repository service / service from CE repository. There are two types of requests: to put a new DS into repository (hereafter s-type) or to get access to a DS which has been already stored in repository (hereafter l-type).

Index t of an item s_t or l_t into random sequence of requests for repository service:

$$R = s_{t=1}^{Ns} \cup l_{t=1}^{Nl} \quad (6)$$

one may take as the time moment when repository receives the request s_t or l_t or, to put it another words arrival time of the next client.

Repository placement policy is mapping:

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

which defines new state of repository (E^*_{RAM}, E^*_{DISK}) given its current state (E_{RAM}, E_{DISK}) and the size, in bytes, of the DS requested. The sequence of repository states is the subset from cartesian product of the E_{RAM}, E_{DISK} or the set of ordered pairs taken from (E_{RAM}, E_{DISK}). The s_t sequence yields pairs (E^*_{RAM}, E^*_{DISK}):

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

The l_t sequence yields pairs:

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

where $E^{(d)}_{RAM}$) - subset of E_{RAM}, comprises DSs moved from cache on disk, $E^{(l)}_{DISK}$ - subset of E_{DISK}, comprises DSs loaded from disk in cache.

To study the l-type requests processing we define exhaustive events: cache hit (HC) – requested DS is in cache, cache miss (MC) – requested DS is not in cache. The object hit ratio (OHR) - the number of HC divided by total number of l-type requests. Let

$$OHR = ohr_{t=1}^{Nl} \quad (8)$$

random sequence or statistics on OHR which has been registered on some time intervals, statistics registered for the subset (6) comprised l-type requests. The integral estimate:

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

is a measure of repository efficiency.

There are two bottlenecks. The first one is processing l-type request in case the request transaction includes operations: DSs-move from cache to disk and DSs-copy from disk to cache (decrease ohr-addends in (8)). The second type of bottlenecks occurs while reading data of a DS stored on disk

(for some reasons repository can not keep it in cache). The latter case is rather obvious: DS size exceeds the repository size limit or there is not enough room in cache to copy the requested DS from disk and all the DSs in cache are still in use by repository clients. Under given circumstances there is no solution to fix the bottleneck at the moment of its emergence. On the other hand one may mitigate the risk of such performance hits, e.g. by minimizing overall time it takes to serve a sequence of l-type requests. Such preventive mechanism can be implemented as a repository placement policy.

We provide another definition of repository placement policy, alternative to (7) in order to pose optimization problem. Recall R - a sequence of request for repository service. Let m - the sequence of corresponding DS-move or DS-copy operations caused by R -requests. Repository placement policy is mapping:

$$h : R \to m,$$

Let functional H is a mapping from cartesian product of h, R to the set of integral estimates $\Sigma(OHR)$ (see (9)):

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

In case of deterministic sequence R one may develop optimal repository placement policy as a solution of the functional H maximization problem:

$$h^* = argmax_h H(h, R),$$

where R - deterministic sequence of requests for repository service.

In general case R is a random sequence and we have to tackle stochastic optimization problem which is much more complicated. To pose such problem we must define the set of invariants that hold for possible input sequences R and choose the optimum criteria. In other words we must clarify which optimal solution h^* we need, e,g. h^* may maximize the expectation $E[\Sigma(OHR)]$ or minimize the variance $Var[\Sigma(OHR)]$.

5.1. REPOSITORY INTERFACE SEMANTICS: FORMAL DEFINITION

We define two axioms which must hold for the most concise repository API.

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

where $\{pre(DS)\}$ - predicate or the formal definition of the precondition "DS instance is well-formed";
$\{post((E^*_{RAM}, E^*_{DISK}))\}$ - postcondition as disjuncture: "repository switched

to a new state, $((E_{RAM} \subset E^*_{RAM})and(E_{DISK} == E^*_{DISK}))or((E_{RAM} == E^*_{RAM})and(E_{DISK} \subset E^*_{DISK}))$" or "repository remains in the previous state, $(E_{RAM} == E^*_{RAM})and(E_{DISK} == E^*_{DISK})$". For the sake of brevity we skip definitions of predicate operands as such definitions in their turn require an alphabet definition. The latter depends heavily on implementations of entities.

DS save-operation must be implementation of copy semantics.

$$\{pre(DS_def)\}$$
$$getData : (E_{RAM}, E_{DISK}, DS_def) \rightarrow$$
$$(E^*_{RAM}, E^*_{DISK}, DS_sc)$$
$$\{post((E^*_{RAM}, E^*_{DISK}, DS_sc))\},$$

where $\{pre(DS)\}$ - predicate or formal definition of precondition "DS_def – a well-formed string identified DS instance by its own metadata or its metadata in repository";
$\{post((E^*_{RAM}, E^*_{DISK}, DS_sc))\}$ - postcondition as disjuncture: "repository switched to the new state, $((E^*_{RAM} == (E_{RAM}$
$E^{(d)}_{RAM}) \cup E^{(l)}_{DISK})and(E_{DISK} \subseteq E^*_{DISK}))$ or remains in the previous state, $((E_{RAM} == E^*_{RAM})and(E_{DISK} == E^*_{DISK}))$, and made shallow copy of DS requested", or "repository remains in the previous state, $((E_{RAM} == E^*_{RAM}, E_{DISK} == E^*_{DISK}))$, requested DS not found in repository".

DS get-operation must be implementation of shallow copy semantics.

6. TWO ARCHITECTURES FOR THE CE DATA REPOSITORY

Since the repository was designed in accordance with the CDB paradigm, its architecture can be described using the following four entities: repository directory (RD), repository cahce logic (RCL), repository cahce (RC), repository persistent storage (RPS).

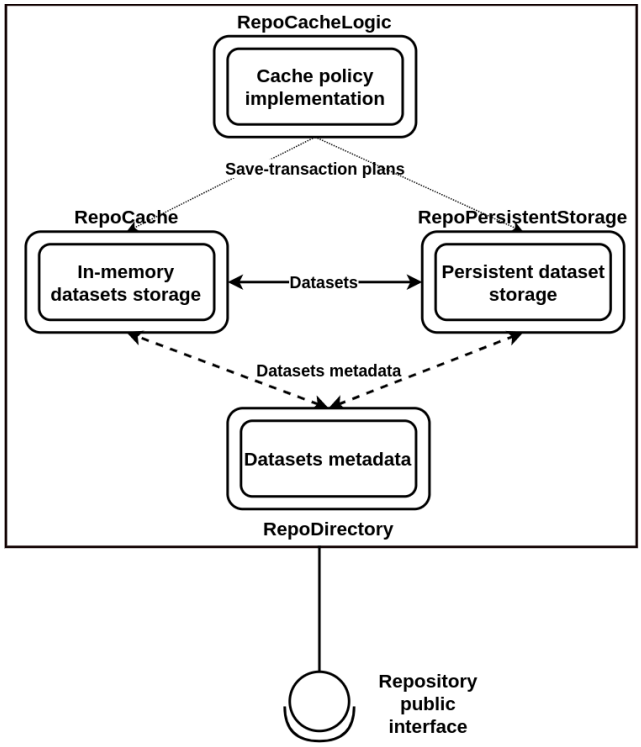

Figure 2: Repository architecture

The procedure of processing the requests of the repository clients (Estimators) is a transaction performed by the instances of all four described entities. The repository implementation coordinates the processing order of operations of transaction. In other words, our implementation of the repository might be considered as a transactional management system developed to handle multidimiensional datasets. The system instance is a singletone within each process (fig. 1) in order to control dataflows between separate components of the CE.

Public intefaces for each of these four components were composed of three operations: runTransaction, commitTransaction, rollbackTransaction. Semantics of the operations wihtin the RCL, RD, RC, RPS is defined by the semantins of the corresponding component, and is assumed to be straightforward to the reader.

Algorithms are presented in concise form, i.e we excluded roll-back op-

erations which recover the state of the repository once a failure of any of its components has happend.

Algorithm 1: Dataset retrieval from the repository

Input : Unique or surrogate dataset key of requested DS in the repository, K

Output: Shallow copy of the requested DS dataset

```
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)
```

Algorithm 2: Dataset placement in the repository

Input : The DS dataset

```
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)
```

7. IMPLEMENTATION-SPECIFIC DETAILS

We were not much lucky to find free software that would meet our repository functional requirements. In order to conduct a comparative analysis of our implementation (hereinafter m-implementation), we also developed its alternative version. It, in our opinion, could be developed by a skillfull programmer, having a formal specification of the public interface (API) of the repository.

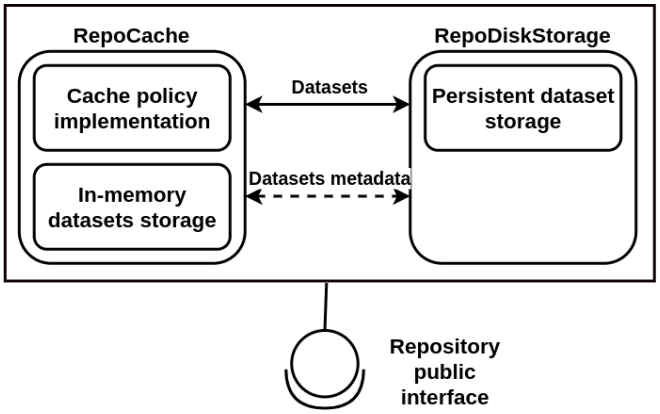

Figure 3: Naive implementation of the repository

An alternative, naive implementation (hereinafter n-implementation) allows us to emphasize the essential features of our software. First, in the n-implementation of the DS entity the data itself will be stored as a one-dimensional array, i.e. without memory fragmentation. However, hardly all DS metadata (tokenss identifying estimator, models, dataset and its individual tracks, as well as arrays of offsets of individual tracks in a DS instance) will be allocated into the single, contiguous block/chunks. In our experience typical implementation do not care of the resulting fragmentation of RAM. Our m-implementation of the DS guarantees that all the data and metadata will be allocated in RAM without fragmentation. Instances of the m-implementation of the DS are copyable and moveable in the sense given by the C ++ language standard for arrays of POD types [14]. Secondly, the possible structure of the n-implementation of the repository code, following the principle of minimalism, is a combination of two components: cache and long-term (persistent) storage, where the cache encapsulates metadata, directory code (i.e. RD) and storage logic (i.e. RCL). Copies of the DS instances accepted for storage by such a repository will be placed in memory according to a naive scenario, i.e. a developer hardly bother to allocate the copy of such instance into contiguous block of RAM. Our m-implementation of the repository code guarantees the placement of copies of the DS in the RC with no heap fragmentation. The very possibility of storing the DS this

way is provided by the above characteristics of the m-implementation of the DS entity (being copyable and moveable). Third, the persistent storage of the n-implementation has been developed using memory-mapped file mechanism, but each DS is stored in a separate file. This option for placing DSs on disk is based on the naive assumption that it significantly reduces the time needed for building serialized DS representations while writing and reading its instance.

According to our estimates, only the first two of these three significant features of the m-implementation reduce the total amount of RAM consumed by the repository by at least one and a half times compared to the n-implementation.

8. METHODOLOGICAL ASPECTS OF THE PERFORMANCE ASSESSMENT

Throughout the application session, we control the part of the process address space used by the RC and RPS components of the storage structure (or rather their containers): the cache and the file-map used for writing DSs on disk (memory-mapped file, mmap). We set upper limits for RAM usage by both the RC and RPS. All four storage components (RCL, RD, RC, RPS) have their own local directories implemented as dynamic arrays. Along CE session the latters accumulate DS metadata, as such their sizes increase. That size grow as a RAM-consumption process can be described as a linear function of the amount of data stored in the repository. That unavoidable overhead guarantees the logarithmic time complexity of the meta-information search operation in local directories.

From the perspective of the performance of the software implementation of our repositories, the following characteristics are of greatest interest:
- the average duration of a transaction to get a shallow copy of DS requested having the DS has been stored in the repository (hereinafter referred to as l-transaction), i.e. the average time to get a shallow copy of DS requested;
- the average duration of a transaction to save a new DS in the repository (hereinafter referred to as s-transaction).

Next, we consider two types of l-transactions, since the requested DS can be either in the cache (RC) or in the presistent storage (RPS). In the latter case, the l-transaction also includes the operation of moving such DSs from RPS to RC.

Here are two important points regarding the estimates of the average duration of the abovementioned three types of transactions:

1. We developed the estimators by interpreting the accumulated statistics as realizations of a nonstationary stohastic discrete-time processes. To describe them, we have chosen regression models, the variable-regressor is the sequential number of the transaction of the corresponding type. In the

case of s-transactions, their sequential number is uniquely defined by the number of DSs accumulated in the repository by that time point. In the case of l-transactions, such a regression can be interpreted in the sense of estimates on the average.

2. Realizations of stohastic process obtained as a result of transaction monitoring, contain a fairly large number of outliers. In such a situation, classical, non-robust estimates of the parameters of regression models in the L_2 metric are unstable. Therefore, we used robust regression analysis methods.

A series of CEs were planned so that during the application session at least 10^4 s-transactions and at least $3 \cdot 10^4$ l-transactions of both types were processed. Sequences of such s- and l-transactions were the responses on sequences of s- and l-requests, in the sense of (6). The type of each element in a s- and l-request sequence is a random variable that was generated by the folllwoing binomial distribution:

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

The value of the θ parameter was chosen in such a way as to provide the aforementioned ratio of the numbers of s- and l-transactions. The described stochastic flow model was an attempt to construct a sequence of random transitions between the queue network states in the sense of (3), given apriori information on the invariants of the sequence (6) is minimal. The DSs generated during such CEs contained three tracks. This fixed number of tracks was actually an estimate of the average number of tracks in the DSs we observe in our typical experiment setups intended for solving statistical analysis problems. The size of each track is a random variable sampled from a uniform distribution, and the boundaries of its support $support(track\_size)$, have been specified as parameters of the CE. The following estimates are result of four series of CE. In each series the support of the measure:

$$sup(track\_size) := U[1, max\_track\_size],$$

was determined by choosing its right boundary, max_track_size, from the set $\{5, 10, 100, 1000, 10000\}$. In other words, the average DS size (in bytes) in each CE series was $\{250, 500, 10^3, 10^4, 10^5\}$ respectively. This set was, again, chosen based on our own experience in solving specific problems. Statistics were collected on machines with Intel CPUs: the Intel i5 series with a clock speed of at least 3 GHz, RAM - DDR2 / DDR3 from 2 to 8 GB, external storage devices with SATA / PATA interfaces, and operating systems selected for testing, - Windows XP/7, Linux. One of the versions of our application is freely available [13]. One may use it to reproduce the estimates presented by us in this paper.

9. SOFTWARE PERFORMANCE ASSESSMENT

For the start we present the performance estimates for the l-transaction. For performance measure we chose 95% tolerance intervals of the average duration of a l-/s-transaction. Hereafter we limited our benchmark to three well-known implementations of the DS storage policy algorithms. Actually our m-implementation is parameterizable by more than a dozen different storage policy algorithms, most of which have been developed by us. For all the implementations a transaction performance estimates are comparable in order of magnitude. In the tables presented below, the average DS size, in bytes, is denoted as S, and the storage policy is denoted as h.

Table 1. Estimates of the boundaries of the 95% tolerance interval for the average duration of the transaction "retrieving a shallow copy from the repository by the unique key of the DS given the DS is cached", µs

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

Table 2. Estimates of the boundaries of the 95% tolerance interval for the average duration of the transaction "retrieving a shallow copy from the repository using a unique key of the DS, provided that the DS has been cached within the previous transaction", μs

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

Table 3. Estimates of the boundaries of the 95% tolerance interval for the average duration of the transaction "retrieving a shallow copy from the repository by the surrogate key of the DS given the DS is cached", μs

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

Table 4. Estimates of the boundaries of the 95% tolerance interval for the average duration of the transaction "retrieving a shallow copy from the repository by the surrogate key of the DS, provided that the DS has been cached within the previous transaction", μs

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

It is apparent that the average duration of l-transactions slightly increases as DS average size increases. In the case of the GDSF policy a transaction takes more time. Such estimates are explained by the fact that GDSF algorithm while processing l-transactions collects statistics and registers the number of requests for DS access. Those operations consume additionally from 2 to 6 μs according to our estimates.

For comparison, we present similar estimates obtained for an alternative, naive storage architecture and its n-implementation.

Table 5. Estimates of the boundaries of the 95% tolerance interval for the average duration of the transaction "retrieving a shallow copy from the repository by unique dataset key of the DS, given the DS is cached" , μs

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

Table 6. Estimates of the boundaries of the 95% tolerance interval for the average duration of the transaction "retrieving a shallow copy from the repository using a unique key of the DS, provided that the DS has been cached within the previous transaction", μs

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

Table 7. Estimates of the boundaries of the 95% tolerance interval for the average duration of the transaction "retrieving a shallow copy from the

repository by surrogate key of the DS, given the dataset is cached", μs

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

Table 8. Estimates of the boundaries of the 95% tolerance interval for the average duration of the transaction "retrieving a shallow copy from the repository by surrogate key of the DS, provided that the DS has been cached within the previous transaction", μs

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

In the case of the n-implementation, the limit of 95% of tolerance intervals is somewhat lower if compared to the one of our m-implementation. The latter i.e. the m-implementation takes extra time (i.e., in addition to copying the shared_ptr) to create the shallow copy's Control Block instance along with that shallow-copy. ControlBlock entity was designed by us for controlling shallow-copy access operations (i.e. to handle client access to data stored in the original DS).

We interpret the above estimates as a justification for the following conclusion: the average duration of l-transactions is determined by a constant that grows slightly as the average size of the DS increases (the latter on a logarithmic scale).

Next, we will consider estimates for l-transactions given the requested DS is stored in the persistent storage (RPS). Table 9 breaks them down for the m-implementation of the storage, table 10 - for the n-implementation.

Table 9. Estimates of the 95% tolerance interval for the average duration of the transaction "retrieving a shallow copy from the repository by a unique key of the DS, given the DS has been cached within the same transaction", μs

h \ S	250	500	10^3
LRU	170;400	110;385	120; 500
MRU	180;325	100;380	100;400
GDSF	175;370	165;310	90;400

Table 9. The second part.

h \ S	10^4	10^5
LRU	$125; 10^3$	$150; 1.2 \cdot 10^3$
MRU	110;950	$130; 1.1 \cdot 10^3$
GDSF	100;2750	$120; 2.3 \cdot 10^3$

In a series of CEs that were carried out to collect those statistics, in some cases we observe the appearance of a linear dependence of the average duration of such a transaction on the number of DSs in the repository. We explain this growth as the degradation of the performance of the system mechanism of memory-mapped files. As the file size increases, the offsets used for building the file-mappings increase as well.

Table 10. Estimates of the boundaries of the 95% tolerance interval for the average duration of the transaction "retrieving a shallow copy from the repository by unique key of DS, given the DS has been cached within the same transaction", μs

h \ S	250	500
LRU	$200; 1.6 \cdot 10^3$	$180; 1.7 \cdot 10^3$
MRU	$190; 10^3$	$180; 1.5 \cdot 10^3$
GDSF	$200; 1.35 \cdot 10^3$	$185; 1.6 \cdot 10^3$

Table 10. The second part.

h \ S	10^3	10^4
LRU	$320; 6.5 \cdot 10^4$	$500; 8.8 \cdot 10^4$
MRU	$220; 4.9 \cdot 10^4$	$500; 8 \cdot 10^4$
GDSF	$300; 7.1 \cdot 10^4$	$480; 8.6 \cdot 10^4$

Table 10. The third part.

h \ S	10^5
LRU	$1460; 9 \cdot 10^4$
MRU	$1490; 7 \cdot 10^4$
GDSF	$1500; 9.2 \cdot 10^4$

It's easy to see that in the case of the naive storage implementation the duration of l-transactions grows significantly as the size of the DS increases.

Estimates of the dynamics of s-transactions demonstrate linear dependencies on the number of DSs in the repository.

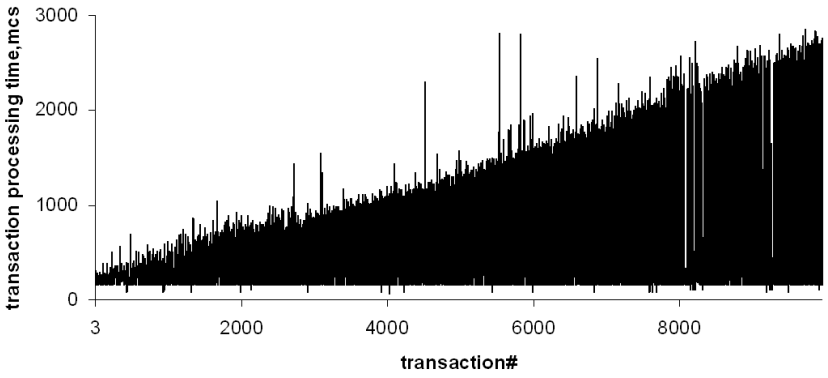

Figure 4: S-transaction duration dynamics. Typical estimates obtained during a single CE

The estimate of the minimum time of s-transactions, which in most series of experiments was obtained when the number of DSs stored in the RPS did not exceed 10^2, is in the range of 200 - 300 μs. The coefficient that determines the linear dependence of the average time of s-transactions on the number of stored DSs is of the order of $2 \cdot 10^{-2} - 3 \cdot 10^{-2}$. That means after the accumulation of more than 10^3 DSs in the RPS file, a significant (by $20 - 30$ μs) increase in the average duration of s-transactions will be observed.

In order to understand the nature of such a phenomenon, it is necessary to take into account the semantics of the s-transaction implemented in the RPS code. RPS begins processing of an s-transaction with its directory look-up: it searches for the metadata of the received DSs in the local (RPS level) directory. If such a DS is registered in the RPS directory, then the s-transaction is accomplished (because the DS is already stored in the RPS). The time complexity of such a search is logarithmic. In our series of CE the duration of that operation grows from several microseconds to two or three tens of microseconds. The main costs, in terms of time, fall on direct copy (memcpy) of DS to an external file mapped into RAM. Such copying is periodically preceded by changing (increasing) the size of the external file along with the sequence (atomic transaction of the RPS level) of two operations: closing the current file-mapping for writing and creating a new file-mapping at a new offset. Dynamic expansion of RPS external file size is another mechanism intended for saving computational resources. The user may specify (in the CE configuration file) the initial size of the RPS file of several tens of gigabytes, thereby completely eliminating the operations of dynamic expansion of that file. The file-mapping for writing allows us to minimize the time it takes to copy the DS to RPS file, but at cost some extra RAM consumption.To minimize the latter we limit the maximum size of that file-mapping and rebuild it quite often. It becomes clear that the average duration of an s-transaction might be the sum of three addends at least: the time which the copy operation takes, the time which RPS file dynamic growth operation takes (a rare event), and the time which the file-mapping rebuild takes (in case DS size exceeded maximum size of the file-mapping). According to the results we have accumulated, it is the last term that determines the linear dependence of s-transactions on the number of DSs in the repository (the total size of DSs stored in the RPS-file at this time point).

Moreover, the rearrangement of file-mappings for writing (according to the statistics obtained as a result of s-transactions monitoring) quite often can be modelled as transient stochastic process in the queue network. Meantime it is not possible to provide an adequate statistical model for such phenomena. The thorough analysis of the mmap/MMF-implementations is required (i.e the level of system libraries). Such an analysis is not of our current priority goals.

10. CONCLUSION

Anticipating possible blames regarding the performance estimates of our software implementations given in the paper, we agree that they do not cover all the options for hardware (HW) configurations on which one may run our CE. However, we note that, firstly, the methodology for constructing a set of such configurations is confined to the set of available HW or the specific

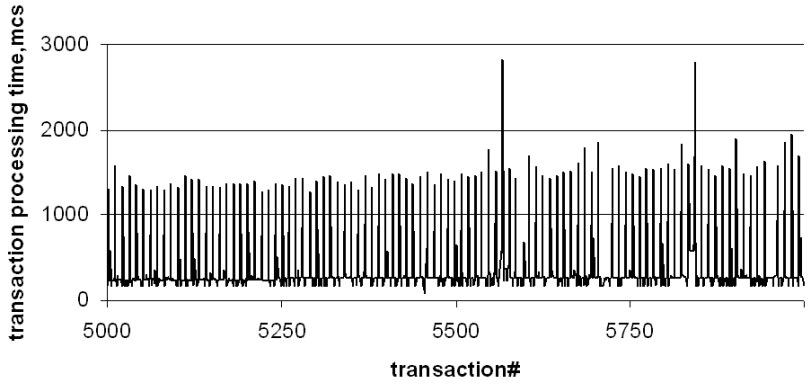

Figure 5: A sample fragment of the realization presented on the fig. 4 illustrating the fine structure of the mentioned stohastic process

subset of the latter supposed to run the CE. Secondly, the performance estimates provided are not specific benchmarks intended for the end user of the software implementation of the repository. We presented our estimates as an experimental justification of the conclusions regarding the time complexity of l-/s-transactions. If the average l-transaction duration is estimated as a constant, on a machine with different hardware one may get a different order of the estimates (given in the latter case the same repostiry instance configuration parameters and the same CE scenario as ours). However, the estimate of the average duration of a l-transaction shall remain constant. By installing SSD drives on the machine its user can reduce the order of the descripancy observed for the two types of l-transactions. But the very nature of these dependencies on the number of DSs in the repository will remain the same.

The reported significant difference in the average durations of the two types of l-transactions (the first type includes the requested DS load-operation from RPS into RC, the second does not as the requested DS is already in the RC) one can not interpret as the flaws of our design.

Recall that the loss of time for transferring DS from RPS to RC is incomparable with the loss of time for accessing the elements of that DS if it cannot be placed in the storage cache and its data are retrieved by sequential read operations from an external storage device. In conclusion, we don't leave the hope that the results of our work are of interest to a fairly wide range of software engineers whose professional activities include solving such problems, and that the inevitable criticism of our results will be constructive.

11. ACKNOWLEDGMENTS

One of the authors would like to express his gratitude to his students and colleagues: Andrei Olegovich Gavrilov and Artyom Sergeevich Tetyukhin, who at different times contributed code related to this work.

The work was our initiative. Neither a legal foundation/entity nor a person support our work (financially or in any other way). We used only free and open-source software/tools (CMake, CodeBlocks, gcc) for implementing the algorithms described in this article.

References

[1] Boost. http://www.boost.org

[2] RaftLib. http://raftlib.io

[3] R-package. http://www.cran.org

[4] *J. Dean et al.* TensorFlow: A system for large-scale machine learning. – Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation, pp. 265–283, 2016

[5] *McIlroy, M.D.* Mass-produced software components. Eds. Naur, Randall. Proceedings, NATO Conference on Software 1969), pp. 88-98, 1969

[6] *Lavrischeva E.M.* Development of domestic programming technology. - Cybernetics and Systems Analysis. 2014. 50, N 3. pp.845–853.(in Russian)

[7] *Lavrischeva E.M., Grischenko V.N.* Assembling programming. Fundamental industry of program systems. – Nauk. dumka, 2009.–p.371 (in Russia)

[8] https://www.ispras.ru/lavrishcheva/

[9] *Ivankov A.A.* The software for real-time investigation of autocontrol mechanisms of transcranial blood circulation. Nauchno-tekhnitcheskiye vedomosti. Physico-matematicheskiye nauki, 2014, 3(201), pp. 92-109.(in Russian)

[10] *Ivankov A.A.* Software platform for real time investigation of cerebral hemodynamics. Humanities and Science University Journal, 2014, 10, p.37-49.

[11] *Szyperski Clemens, Gruntz Dominik, Murer Stephan* Component Software Beyond Object-Oriented Programming. Second Edition. Addison-Wesley, p. 589, 2002

[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, paragraph "Trivial move constructor"

12. Addendum 1. An example of a CE configuration file fragment.

```
# inyerval estimate of Poisson model paremeter
STEP_2 = task_cfg/POISSONLAMBDA/taskStep2.cfg
```

13. Addendum 2. The contents of the task_cfg/POISSONLAMBDA/taskStep2.cfg. configuration file

```
# estimator input data definition (using domain specific language)
REPO_ENTRIES = [ 1 [ 0 ] ]; [2 [0]]

    # estimator type definition
ESTIMATOR_TYPE = INTERVAL_ESTIMATOR

    # string literal defining estimator implementation to use
ESTIMATOR_NAME = IntervalPoissonLambda

    # path to the estimator configuration file
ESTIMATOR_PARAMS_SRC = task_cfg/POISSONLAMBDA/IntervalPoissonLambda.cfg

    # declarative definition of class of the model used for the parametriza-
tion of the estimator
MODEL_INTERFACE_TYPE =ProbabilisticModelCompositionInterface

    # declarative definition of the model used for the parametrization of the
estimator
MODEL_NAME = POISSON_MODEL
```