DOI 10.37882/2223–2966.2021.08.09

РАЗРАБОТКА СИСТЕМЫ ХРАНЕНИЯ С ИСПОЛЬЗОВАНИЕМ МЕТОДОВ БЫСТРОГО ДОСТУПА К ДАННЫМ

DEVELOPMENT OF A STORAGE SYSTEM USING QUICK DATA ACCESS METHODS

N. Ermakov
S. Molodyakov

Summary. This paper discusses the issue of building applications that provide acceleration of data access. A method for caching database queries is proposed, which requires preliminary analysis of the query. The developed data storage system is described. The system differs from the known ones in that it simultaneously uses a database, caching, file storage system and a search engine. The results of testing the caching efficiency are presented.

Keywords: database, file system, caching, search engine, application server.

Ермаков Николай Вадимович
Аспирант, Санкт-Петербургский политехнический университет Петра Великого
kolyaermakov@yandex.ru
Молодяков Сергей Александрович
Д.т.н., профессор, Санкт-Петербургский политехнический университет Петра Великого
samolodyakov@mail.ru

Аннотация. В данной работе рассматривается вопрос построения приложений, которые обеспечивают ускорение доступа к данным. Предложен метод кэширования запросов к базе данных, который предусматривает предварительный анализ запроса. Описана разработанная система хранения данных. Система отличается от известных тем, что в ней одновременно используется база данных, кэширование, файловая система хранения данных и поисковый движок. Приведены результаты тестирования эффективности кэширования.

Ключевые слова: база данных, файловая система, кэширование, поисковый движок, сервер приложений.

Введение

Во многих областях собирается большое количество данных. Например, машинное обучение в основном хорошо работает только когда у нас есть большие массивы данных [1]. Однако рост данных ставит новые вопросы и задачи перед парадигмой управления данными [2]. Поскольку увеличилось количество данных, процесс управления данными также должен измениться.

Система хранения данных должна хранить большое количество информации, сохранять её при любых обстоятельствах, предоставлять к ней постоянный доступ и не терять данные. Важными параметрами системы являются безопасность, отказоустойчивость [3] и производительность.

Работать с большим количеством данных нелегко. Необходимо найти место для хранения вновь собранных данных. Однако расширение емкости системы хранения данных — не единственная проблема. Огромный рост данных и хранилищ сопровождается сложностями при управлении большим архивом данных. По мере роста архива становится все труднее находить, какие элементы данных необходимы, переносить данные из одной технологии в другую, реорганизовывать при изменении данных и целей, а также устранять сбои оборудования. Другими словами, сложность проблемы растет в геометрической прогрессии [4].

Другая проблема заключается в том, что данные не являются статичными, они часто меняются. Если данные редко читаются, то простое расширение емкости системы хранения имеет смысл. Однако в часто это не так. Данные бесполезны без совместного использования, анализа и обработки для подтверждения новой гипотезы или оценки новых алгоритмов. Если все данные хранятся в одном месте и доступны для всех, то попытки большого количества людей одновременно получить доступ к традиционной системе хранения могут приводить к отказу системы. Система хранения может стать узким местом, которое замедляет всех. Быстрое решение заключается в репликации и хране-

нии данных в нескольких местах, что повышает доступность для пользователей. Однако хранение нескольких копий одних и тех же данных увеличивает сложность системы хранения. Необходимо отслеживать местоположение каждой реплики и обновлять каждую копию при внесении изменений.

Данные обычно содержат связанные метаданные, которые описывают различные атрибуты данных [5, 6]. Они обычно включают в себя традиционные метаданные файловой системы, такие как размер файла, права доступа, время создания, а также информацию более высокого уровня, такую как данные измерений или используемых инструментов, информацию об изучаемых предметах или объектах и другие специфичные для предметной области знания, которые исследователи считают полезными. Проблема возникает, когда у вас есть не десятки или сотни, а десятки тысяч и миллионы объектов данных. Вы хотите иметь возможность хранить и обмениваться данными с вашими коллегами. Вы хотите запросить данные с определенными ограничениями на метаданные. Однако по мере того, как данные растут, задача поиска правильного набора данных становится все труднее. Если набор данных становится слишком большим, невозможно выполнить поиск необходимых данных путем сканирования всего хранилища. Очевидно, что существует потребность в системе хранения, которая поддерживает хранение метаданных и поиск по ним. Система должна не только уметь хранить много данных, но и быстро искать по ним.

Типовым подходом в решении данной задачи является распараллеливание метаданных, расположение на разных серверах [7, 8]. Недостатками такого подхода являются задержки в сетях коммуникации, необходимость поддержания работоспособности большого количества серверов и контролирования доступа к каждому серверу, что приводит к дополнительным потерям скорости и вызывает дополнительные сложности в администрировании, что приводит к уменьшению экономической эффективности.

Две канонические модели для хранения данных — файловая система и база данных — не очень подходят для долгосрочного хранения данных. Реляционная база данных хорошо подходит для запросов поиска и сортировки, но она не оптимизирована для хранения больших двоичных объектов. С другой стороны, файловая система хорошо подходит для простого размещения больших двоичных объектов по мере их создания. Тем не менее, по мере того как файловая система становится больше, сортировка и поиск данных выполняются менее эффективно. По мере роста количества данных файловая система не может удовлетворить все потребности. Таким образом, в файловых системах отсутствуют возможности, необходимые для эффективного поиска по данным.

Данные также могут храниться в поисковом движке. В поисковом движке часто отсутствует поддержка транзакций, а данные хранятся в денормализованном виде. Денормализация повышает производительность извлечения (поскольку не требуется объединять данных из разных таблиц), использует больше места (поскольку некоторые данные должны храниться несколько раз), но затрудняет поддержание согласованности и актуальности (так как любое изменение должно применяться ко всем экземплярам). Тем не менее, поисковый движок отлично подходит для данных, которые записали один раз и затем много читают. Примеры поисковых движков — Elasticsearch и Solr, которые основаны на Lucene [8].

Для ускорения поиска данных в базах данных и поисковых движках используются индексы. Индексирование увеличивает время добавления или изменения, поскольку при этом приходится обновлять сами индексы, но существенно сокращает время поиска. Базы данных используют индекс на основе B-дерева, поисковые движки в основном использует инвертированный индекс [10].

Кэширование также является одним из способов ускорить поиск данных. При кэшировании часто запрашиваемые данные помещаются в более быструю, но меньшую по объему память. Для определения оптимальных параметров кэширования может применятся моделирование [11].

Существуют системы хранения данных, в которых используется несколько технологий. Например, данные могут храниться одновременно в базе данных и поисковом движке. Существуют также системы хранения данных, состоящие из базы данных и файловой системы.

Ниже будет рассмотрено как применение методов кэширования при работе с данными, так и использование файловых систем и поисковых движков. Ожидается, что время доступа к данным в основном уменьшится из-за кэширования результатов запросов к базе данным.

Вклад этой статьи заключается в следующем: 1) дизайн системы хранения данных с многопользовательским доступом, который использует разные технологии для разных типов данных. 2) разработанный метод кэширования запросов к базе данных, который требует декомпозицию запроса.

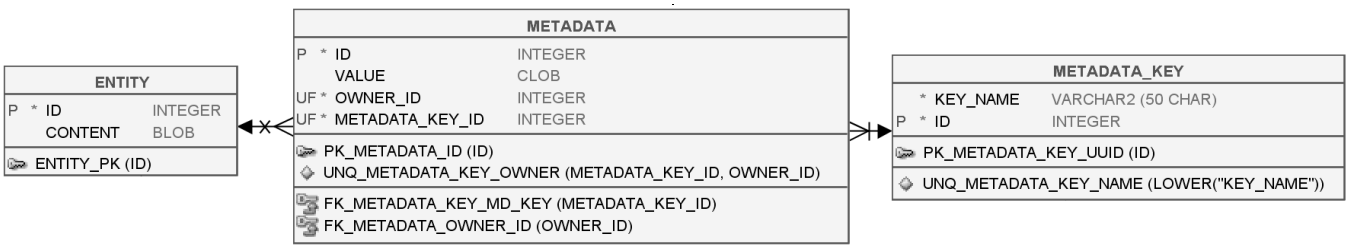

Рис. 1. ER диаграмма метаданных

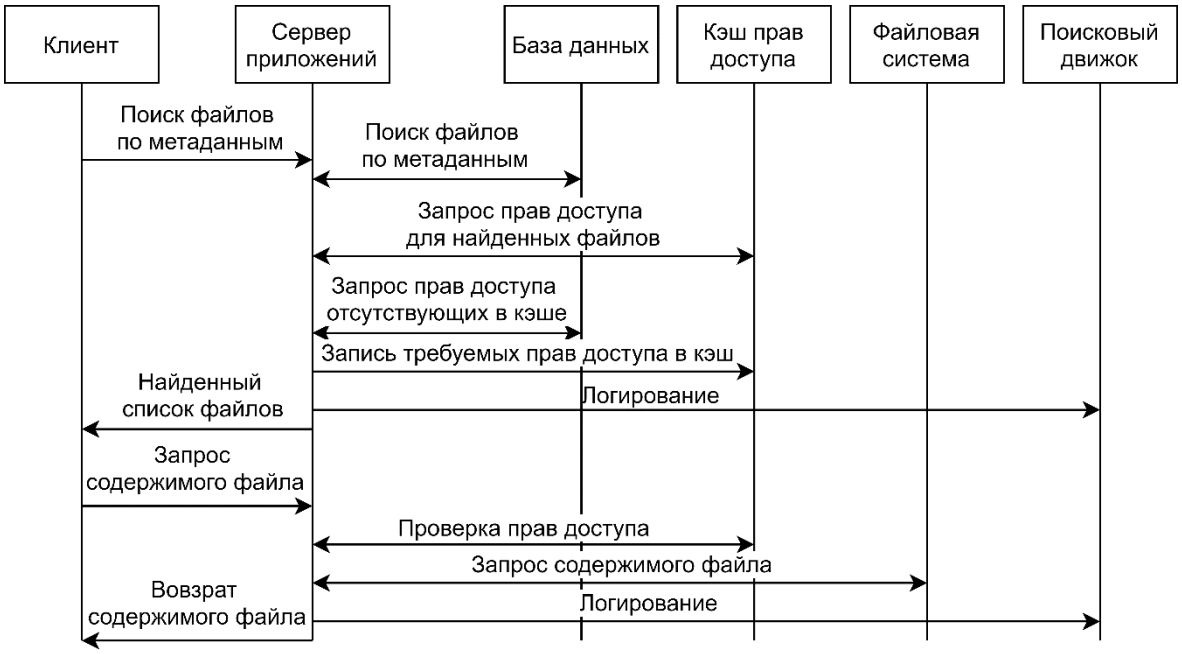

Рис. 2. Диаграмма работы системы при поиске по метаданным

Разработанная система хранения данных

Особенности разработанной системы:

1. Можно задать права доступа для каждого пользователя и для каждого файла и папки. Права доступа могут помещаться в кэш.
2. Каждый файл и каждая папка могут иметь пользовательские метаданные. Система позволяет осуществлять произвольный поиск файлов и папок по их метаданным.
3. Используется поисковый движок для быстрого поиска по событиям, произошедших в системе.

Система хранения состоит из файловой системы, реляционной базы данных, распределенного кэша с правами доступа, готовой распределенной поисковой системы для хранения журналов активности, специального приложения, которое взаимодействует со всеми соответствующими компонентами. В базе данных хранятся метаданные файлов (например, размер, кем и когда создан и изменен, имя, путь, другие пользовательские метаданные). Возможно использовать произвольные ключи для хранения метаданных. Имя ключа хранится в таблице Metadata_key, а значение хранится в таблице Metadata (рисунок 1). У одного ключа может быть несколько значений.

На каждый файл каждому пользователю или группе пользователей могут быть выданы права на чтение, запись и удаление. Права доступа часто запрашиваются и редко меняются, поэтому может быть использован кэш. Если клиент посылает запрос на обновление кэшированных данных, то сервер приложений обновляет их и в кэше, и в базе данных. Все произошедшие в системе события (создание, изменение, удаление файлов или метаданных) записываются в поисковый движок Elasticsearch.

Клиент запрашивает у сервера приложений некоторую информацию (метаданные, файлы или произошедшие события), а сервер приложений обращается

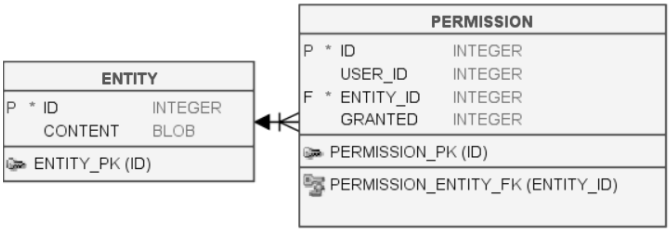

Рис. 3. Пример ER диаграммы базы данных № 1.

к нужным подсистемам для выдачи нужного результата. На рисунке 2 приведена диаграмма работы системы, когда пользователь хочет найти файлы по метаданным.

Приложение было написано на языке Java с использованием фреймворка Spring. В качестве сервера приложений использовался Apache Tomcat. Был выбран кэш ehCache, так как его удобно использовать вместе со Spring. Hibernate и Spring Data использовались для доступа к базе данных. Для доступа к Elasticsearch использовался JEST Client.

Использовалась база данных Oracle. Клиенты обращаются к приложению по HTTP протоколу (REST API). Несколько клиентов могут одновременно посылать запросы к системе. Мы используем транзакции для параллельного выполнения запросов в базе данных. Настройки кэша описываются в xml формате. Можно указать время жизни кэшированных данных или что данные не удаляются по времени, максимальное количество данных в памяти, хранить ли данные на диске, политику удаления данных при нехватке памяти (может быть LRU — least recently used, LFU — least frequently used, FIFO — first in first out, the oldest element by creation time).

```
<cache name="cachename"
eternal="true"
maxElementsInMemory="500000"
overflowToDisk="false"
diskPersistent="false"
memoryStoreEvictionPolicy=»LFU»/>
```

Java метод, который запрашивает права доступа для заданного пользователя и сущности, аннотирован аннотацией @Cacheable. Разрешения запрашиваются из кэша, если они там есть. Если разрешения отсутствуют в кэше, то они запрашиваются из базы, после чего автоматически кладутся в кэш. Хеш код входных параметров метода (userId и entityId) является ключом в кэше. Выходной параметр метода (granted) является значением. Аннотации @CacheEvict и @CachePut используются для очищения и обновления кэша. Сущности фильтруются с помощью аспектно-ориентированного программирования после чтения из базы. Уже упомянутый метод с аннотацией @Cacheable вызывается для каждой полученной сущности.

Метод кэширования запросов

Запросы часто выбирают данные на основе нескольких таблиц базы данных. В данной статье рассмотрим схему, при которой кэшируются не все таблицы, участвующие в запросе. Эта схема может применяться, если некоторые таблицы базы данных не могут быть кэшированы, так как, например, эти данные часто меняются или требуют много памяти.

Запрос в базу разбивается на два запроса. Первый запрос не обращается к кэшированным таблицам, а второй выполняется, только если результат отсутствует в кэше. Также должна быть добавлена обработка полученных данных на уровне сервера приложений. Рассмотрим на примере (рисунок 3). Похожий запрос может выполняться в разработанной системе.

Пусть в базе данных хранятся некоторые сущности (таблица Entity) и на каждую сущность проставлены права доступа для каждого пользователя (таблица Permission). Пусть таблица Entity не может быть кэширована. Права доступа на сущность могут быть помещены в кэш.

Допустим, мы хотим выбрать все сущности, на которые есть права у заданного пользователя. Пусть поле granted равно 1, если права есть. Запрос будет выглядеть следующим образом:

```
SELECT DISTINCT e.* FROM entity e
JOIN permission p ON e.id = p.entity_id
WHERE p.user_id =: userId AND granted = 1;
```

Будем хранить все права доступа в кэше по составному ключу из полей user_id и entity_id, значением будет поле granted. Для заполнения кэша будем использовать запрос **SELECT** * **FROM** permission **WHERE** user_id =: userId **AND** entity_id =: entityId. Получать из базы все сущности можно с помощью запроса **SELECT** * **FROM** entity. Из полученных сущностей не-

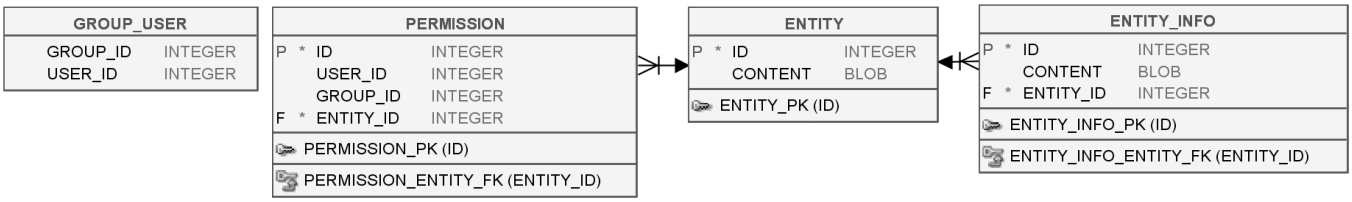

Рис. 4. Пример ER диаграммы базы данных № 2.

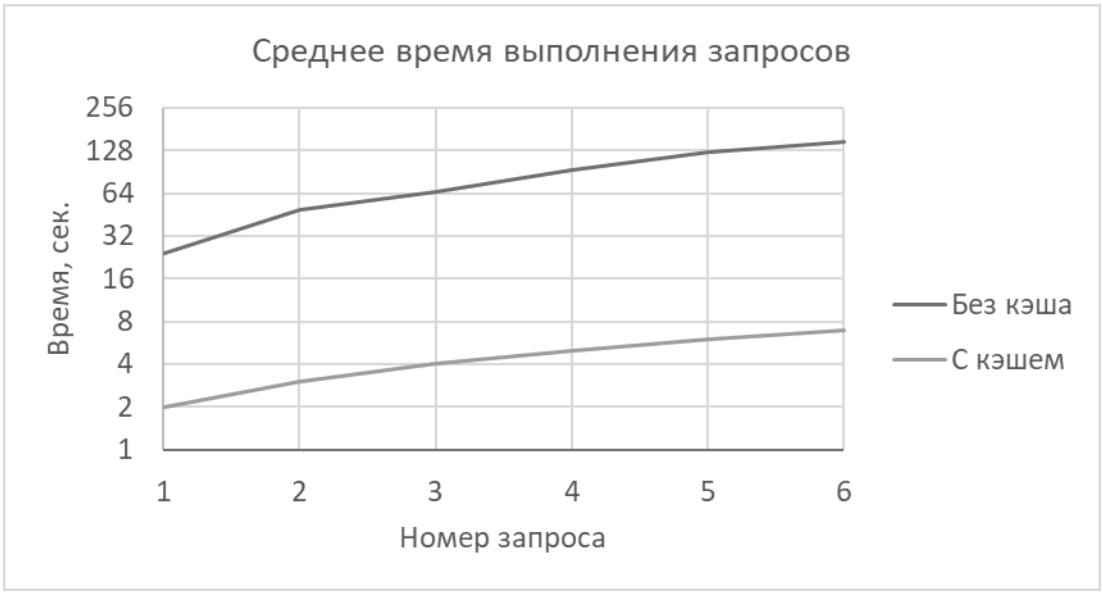

Рис. 5. Время выполнения запросов

обходимо оставить только те, на которые у данного пользователя есть права. Мы можем это сделать, используя кэш.

Рассмотрим другой пример (рисунок 4). По сравнению с предыдущим примером добавились группы (таблица Group_user) и дополнительная информация о сущности (таблица Entity_info). Если группе даны права на сущность, то пользователи, состоящие в этой группе, имеют права на сущность. Таблицы Entity_info и Entity не кэшируются, таблицы Permission и Group_user кэшируются. Запрос будет выглядеть следующим образом:

```
SELECT DISTINCT e.* FROM entity e
JOIN permission p ON e.id = p.entity_id
JOIN entity_info ei ON e.id = ei.entity_id
JOIN group_user gu ON p.group_id = gu.group_id
WHERE (p.user_id =: userId OR gu.user_id =: userId)
AND granted = 1;
```

Необходимо разбить этот запрос на два. Первый запрос будет каждый раз обращаться в базу:

```
SELECT * FROM entity e
JOIN entity_info ei ON e.id = ei.entity_id
```

Результаты второго запроса будут класться в кэш:

```
SELECT * FROM permission p
JOIN group_user gu ON p.group_id = gu.group_id
WHERE (p.user_id =: userId OR gu.user_id =: userId)
AND p.entity_id =: entityId
```

Затем происходит фильтрация полученных сущностей с использованием кэша.

Если в кэшированных таблицах происходят изменения, то необходимо очищать, либо обновлять кэш.

Тестирование эффективности кэширования

Система хранения данных была развернута для тестирования на локальном компьютере с HDD. На рисунке 5 приведено среднее время выполнения нескольких запросов без кэширования и с кэшированием.

Запросы осуществляют сложные поиски по метаданным. Все запросы разные, но имеют общую часть — фильтрацию на основе прав доступа. Таблица в базе данных с правами доступа содержит около 5 миллионов записей для 130 тысяч сущностей и 100 пользователей. Размер базы данных — около 5 Гб. Запросы возвращают 100–1000 сущностей. Видно, что кэширование существенно сокращает время выполнения запросов. Ниже приведен SQL запрос № 6 из таблицы без кэширования:

```
SELECT DISTINCT e3.* FROM entity e1
LEFT JOIN metadata m1 ON e1.id=m1.owner_id
LEFT JOIN metadata_key mk1 ON m1.metadata_key_
id=mk1.id
CROSS JOIN entity e2
LEFT JOIN metadata m2 ON e2.id=m2.owner_id
LEFT JOIN metadata_key mk2 ON m2.metadata_key_
id=mk2.id
CROSS JOIN entity e3
LEFT JOIN metadata m3 on e3.id=m3.owner_id
LEFT JOIN metadata_key mk3 on m3.metadata_key_
id=mk3.id
LEFT JOIN permission p ON p.entity_id = e3.id
LEFT JOIN group_user gu ON p.group_id = gu.group_id
WHERE
p.granted = 1 AND (p.user_id = 'user' or gu.user_id =
'user') AND
mk3.key_name='k3' AND LOWER(m3.value) LIKE 'v31'
AND mk2.key_name='k2'
AND (LOWER(m2.value) LIKE 'v21' OR LOWER(m2.
value) LIKE 'v22'
OR LOWER(m2.value) LIKE 'v23' OR LOWER(m2.value)
LIKE 'v24'
OR LOWER(m2.value) LIKE 'v25' OR LOWER(m2.value)
LIKE 'v26')
AND (e1.id NOT IN
(SELECT e4.id FROM entity e4
INNER join metadata m4 ON e4.id=m4.owner_id
INNER join metadata_key mk4 ON m4.metadata_key_
id=mk4.id
WHERE mk4.key_name='k1')
OR mk1.key_name='k1' AND LOWER(m1.value) LIKE
'v1')
AND e1.id=e2.id AND e2.id=e3.id;
```

При наличии кэширования в этом запросе отсутствуют два оператора join с таблицами Permission и Group_user и условие в операторе where, проверяющее права доступа. Права доступа находятся в кэше и при необходимости загружаются из базы. Этот запрос осуществляет поиск по 6 значениям v21, …, v26 у ключа k2. В запросах № 1–5 поиск идет соответственно по 1–5 значениям.

Заключение

Была разработана система хранения данных, которая умеет эффективно хранить данные в зависимости от их особенностей. Для этого необходимо применять различные технологии. Неструктурированные данные хранятся в файловой системе, структурированные — в базе данных, не меняющиеся — в поисковом движке, а часто запрашиваемые данные кэшируются.

Была предложена схема кэширования, при которой кэшируется часть таблиц, участвующих в запросе. Запрос в базу упрощается, не нужно делать трудоемкие операции JOIN с кэшированными таблицами. К сожалению, в этом случае из базы передаются лишние данные, которые затем будут отфильтрованы с использованием кэша. Таким образом, возрастает нагрузка на сервер приложений и увеличивается количество передаваемых данных между базой и сервером приложений. Нагрузка на базу данных, как правило, уменьшается. Это обычно и требуется, так как нагрузка на реляционную базу данных плохо распараллеливается. Эффективность предложенной схемы кэширования зависит от данных, частоты обновление кэша, характеристик серверов и сети. В некоторых случаях производительность может ухудшиться.

ЛИТЕРАТУРА

1. Монастырев В. В., Молодяков С.А. Применение методов машинного обучения для анализа интересов пользователей // Современная наука: актуальные проблемы теории и практики. Серия: Естественные и Технические Науки. 2021. № 01. С. 97–101.
2. Big data: From beginning to future / Yaqoob I. [и др.] // International Journal of Information Management. 2016. Т. 36, № 6. С. 1231–1247.
3. Development of tools for improving the data storage systems reliability as a part of digital transformation strategy / Bolsunovskaya M.V. [и др.] // IOP Conference Series: Materials Science and Engineering. IOP Publishing, 2020. Т. 940.
4. Big Data technologies: A survey / Oussous A. [и др.] // Journal of King Saud University-Computer and Information Sciences. 2018. Т. 30, № . 4. С. 431–448.
5. Metadata management for scientific databases / Pinoli P. [и др.] // Information Systems. 2019. Т. 81. С. 1–20.
6. Thomson A., Abadi D.J. Calvinfs: Consistent WAN replication and scalable metadata management for distributed file systems // 13th USENIX Conference on File and Storage Technologies (FAST 15). 2015. С. 1–14.
7. Hopsfs: Scaling hierarchical file system metadata using newsql databases / Niazi S. [и др.] // 15th USENIX Conference on File and Storage Technologies (FAST 17). 2017. С. 89–104.

8. IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion / Ren K. [и др.] // SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2014. С. 237–248.
9. Berryman J., Turnbull D. Relevant search: with applications for Solr and Elasticsearch. Simon and Schuster, 2016.
10. Performance evaluation of inverted files, B-Tree and B+ Tree indexing algorithm on Malay text / Rosnan S. [и др.] // 2019 4th International Conference and Workshops on Recent Advances and Innovations in Engineering (ICRAIE). IEEE, 2019. С. 1–6.
11. Ermakov N. V., Molodyakov S.A. A caching model for a quick file access system // Journal of Physics: Conference Series. IOP Publishing, 2021. Т. 1864. № . 1.

Санкт-Петербургский политехнический университет Петра Великого