https://doi.org/10.31891/2219-9365-2023-74-18
УДК 004.7

ОСТРОВСЬКИЙ Денис
Хмельницький національний університет
https://orcid.org/0009-0002-1747-9444
e-mail: azatot2014@gmail.com
ЛИСИЙ Андрій
Хмельницький національний університет
https://orcid.org/0009-0001-0065-9740
e-mail: andrii.lysyi1@gmail.com
СВИСТУН Сергій
Хмельницький національний університет
https://orcid.org/0009-0009-8210-6450
e-mail: svystuns@khmnu.edu.ua
ОНИШКО Оксана
Хмельницький національний університет
https://orcid.org/0000-0002-2125-4160
e-mail: van4o@ukr.net
СЕРГЄЄВ Євгеній
Хмельницький національний університет
https://orcid.org/0009-0008-9877-9863
e-mail: ysierhieiev@gmail.com

АРХІТЕКТУРА СХОВИЩА МАСИВІВ З КОМПАКТНИМ ІНТЕГРОВАНИМ ІНДЕКСОМ

Збільшення кількості даних, що генеруються щодня потребує ефективного їх зберігання, швидкого запиту до таких даних. Як правило, такі дані є багатовимірними і можуть бути представлені за допомогою моделі даних масиву. Поряд з додаванням в систему все більш і більш потужних процесорів і прискорювачів, більшість сучасних обчислювальних систем містять все більш складний стек вводу-виводу, починаючи від традиційних дискових файлових систем і закінчуючи гетерогенними прискорювачами з індивідуальним простором пам'яті. Ефективний доступ до такого складного стека вводу-виводу при обробці масивів має важливе значення для використання великої обчислювальної потужності сучасних обчислювальних платформ. Одним із ключів до досягнення такої ефективності є визначення місця генерації або зберігання даних, а також відповідний вибір відповідних стратегій представлення та обробки.

В цій роботі зосереджено на оптимізації обробки масивів у таких складних стеках вводу-виводу шляхом дослідження двох фундаментальних питань: яке представлення даних слід використовувати, і де дані повинні зберігатися та оброблятися. Таким чином, розглянуто проблему ефективної обробки даних масиву, представлено компактне сховище масивів для дискових даних, інтегруючи в нього індексацію на основі значень без втрат.

Розроблена архітектура системи зберігання масивів з інтегрованою підтримкою індексу вартості. Завдяки їй досягається реорганізація елементів в ряд користувацьких бітів і ефективне кодування індексів згідно бітів і відповідних їм значень. При цьому генерується індексоване представлення масиву, яке додає мало додаткових витрат на зберігання.

Напрямами подальших досліджень є удосконалення архітектури системи зберігання масивів з інтегрованою підтримкою індексу вартості в частині оптимізації зберігання індексів.

Проведені експерименти підтверджують можливість практичної реалізації запропонованих архітектурних рішень.

Ключові слова: сховище масивів, компактні інтегровані індекси, архітектура.

OSTROVSKY Denis, LYSYI Andriy,
SVISTUN Sergey, ONYSHKO Oksana, SERGEYEV Yevgeny
Khmelnitskyi National University

ARRAY STORAGE ARCHITECTURE WITH COMPACT INTEGRATED INDEX

Increasing the amount of data generated daily requires effective storage, a quick request for such data. Typically, such data are multidimensional and can be presented using the array data model. Along with the addition of more and more powerful processors and accelerators, most modern computing systems contain an increasingly complex input stack, ranging from traditional disk file systems and ending with heterogeneous accelerators with individual memory space. Effective access to such a complex input stack when processing arrays is important for the use of high computing capacity of modern computing platforms. One of the key to achieving such efficiency is to determine the place of generation or storage of data, as well as the corresponding selection of appropriate representation and processing strategies.

In this work, it is focused on optimizing the processing of arrays in such complex stacks of input-output by studying two fundamental questions: what kind of data should be used, and where the data should be stored and processed. Thus, the problem of efficient processing of data of the array is considered, a compact storage facility for disk data is presented, integrating it in its indexation based on values without loss.

The architecture of the storage system of arrays with integrated support of the value index has been developed. It achieves the reorganization of the elements into a series of custom bits and the effective coding of indices according to the bits and the corresponding values. This generates an indexed presentation of the array, which adds little additional storage costs.

The areas of further research are to improve the architecture of the storage system of arrays with integrated maintenance of the value index regarding the optimization of storage of indexes.
The experiments confirm the possibility of practical implementation of the proposed architectural decisions.
Keywords: array repositories, compact integrated indices, architecture.

Постановка проблеми у загальному вигляді
та її зв'язок із важливими науковими чи практичними завданнями

Із збільшенням кількості даних, що генеруються щодня, ефективне зберігання та запит таких даних, зазвичай багатовимірних, які можуть бути представлені за допомогою моделі даних масиву, стає все більш важливим. Поряд з додаванням в систему все більш і більш потужних процесорів і прискорювачів, більшість сучасних обчислювальних систем містять все більш складний стек вводу-виводу, починаючи від традиційних дискових файлових систем і закінчуючи гетерогенними прискорювачами з індивідуальним простором пам'яті. Ефективний доступ до такого складного стека вводу-виводу при обробці масивів має важливе значення для використання великої обчислювальної потужності сучасних обчислювальних платформ. Одним із ключів до досягнення такої ефективності є визначення місця генерації або зберігання даних, а також відповідний вибір відповідних стратегій представлення та обробки. Ця робота зосереджена на оптимізації обробки масивів у таких складних стеках вводу-виводу шляхом дослідження двох фундаментальних питань: яке представлення даних слід використовувати, і де дані повинні зберігатися та оброблятися. Тому, розглядається проблема ефективної обробки даних масиву, представлено компактне сховище масивів для дискових даних, інтегруючи в нього індексацію на основі значень без втрат.

Таким чином, розробка архітектури сховища масивів з компактним інтегрованим індексом, є актуальною науковою задачею і її результати можуть бути використані при проєктуванні систем із сховищами масивів різного призначення.

Аналіз відомих рішень щодо архітектури сховища масивів

Використання великих даних і глибоких нейронних мереж надало дослідникам можливість до ефективного отримування та обробки значного великого обсягу даних [1]. Ця тенденція очевидна з обсягу даних, що збираються великомасштабними дослідницькими проектами, починаючи від терабайт великомасштабних зображень телескопів [2] до трильйонів частинок при моделюванні [3-5], від реконструкції медичних зображень [6] для відстеження змін глобального клімату. Більші та швидші системи будуються для задоволення такого попиту на обробку даних. Такі системи мають обчислювальні потужності від одного до двох ексафлопс [6, 7]. Більшість даних, що збираються, мають багатовимірний характер, що робить модель даних *масиву* її природним представленням. У такій моделі даних дані представляються в багатовимірних масивах, пов'язуючи кожен кортеж даних з декількома стовпцями (*атрибутами*) з одним n-вимірним вектором (*координатами*). Традиційно такі дані зберігаються в паралельних файлових системах [8], і легко покладаються на формати даних масивів, такі як доступ. З*бір* даних є лише першим кроком. Зібрані дані повинні бути ефективно оброблені або проаналізовані для створення цінності для суспільства. Аналітика даних масиву, як правило, є онлайн аналітичною обробкою [9]. Запити стилю, починаючи від простої фільтрації, агрегації та об'єднання [10-15] до більш складних запитів, таких як подібність об'єднання та контрастний набір видобутку, в цьому контексті є важливими. Швидкість, або час до розуміння, є важливими у науці, керованій даними, оскільки швидший час до розуміння призводить до швидшого зворотного зв'язку та коригувань, що може значно прискорити загальна ефективність досліджень і розробок. Однак останні архітектурні тенденції представляють значні виклики для такого завдання.

Модель та архітектура сховища масивів з компактним інтегрованим індексом

Задана область значень методів зв'язування може бути великою. Розглядатимемо тільки біти на основі діапазону, тобто, якщо набір даних має домен значення, таким чином розділяючи дані на m бітів. Для ефективної операції підмножини набір даних все ще ділиться на n гіперпрямокутних сегментів однакового розміру відповідно до його розмірів за допомогою регулярного поділу. У поєднанні з m бітами це створює загальну кількість фрагментів. Бітів можуть бути сотні, а деякі біти можуть містити лише кілька елементів, то система групує ці фрагменти разом на множині для кращої ефективності вводу-виводу. Однак, зберігання всіх m бітів сегмента всередині фрагмента призводить до того, що будь-який вибір на основі значень завантажує всі біти з диска, ігноруючи сенс індексації. Отже, можна згрупувати кілька бітів разом, так що кожен поділ містить дані k бітів у сегменті. Індексований поділ можна однозначно ідентифікувати за його граничними координатами та біт, який він містить. Карта фрагментів згідно дерева зберігає адресу кожного фрагмента. Оскільки кількість множин зазвичай не дуже велика, тому система зберігає цю карту в основній пам'яті.

Індексований поділ містить три сегменти: залишкові дані та позиційний індекс, а також заголовок фрагмента для цілей обліку. Одним з питань є розташування сховища позиційних індексів та даних

всередині множини. Проектне рішення виходить із спостереження, що позиційні індекси не завжди необхідні для обробки запитів. Наприклад, оператор агрегації може вибрати всі елементи всередині множини, і, таким чином, не потребує позиційних індексів для продовження. Таким чином, з точки зору компонування, позиційні індекси та залишкові дані всіх бітів у блоці об'єднуються у два окремих сегменти. Коли позиційний індекс не потрібен, завантажується лише залишковий сегмент даних, не шкодуючи зайвого вводу-виводу для завантаження позиційних індексів. Позиційний індекс для кожного елемента в підмножині позначає його розмірні позиції всередині сегмента. Щоб досягти цього, багатовимірні координати елементів спочатку переводяться до лінійних зміщень. Зміщення можуть бути збережені в стислому вигляді, або у вигляді растрового зображення, або з використанням схеми стиснення з перевернутими списками монолітної послідовності. Обидва способи мають свої переваги. Растрові зображення, як правило, більш ефективні у виконанні наближених запитів згідно встановлених перетинів. Однак декодування зміщень з растрового зображення вимагає відстеження того, скільки бітів вже було відвідано, тому призводить до додаткових витрат процесора. На обсяг зберігання двох методів може впливати набір даних і стратегія прив'язки. Залишкові дані підмножини зберігають додаткові дані, необхідні для розрізнення різних значень в одній підмножині. Це можна розглядати як проблему стиснення: як без втрат стиснути значення в підмножині з урахуванням його діапазону. Просте застосування загальних алгоритмів стиснення даних до підмножини значень часто є неоптимальним, оскільки компресор не знає про зв'язок значень. Представимо загальну структуру стиснення даних, обмежених діапазоном. Стискаємо дані за два кроки: зіставляємо і кодуємо. Фаза карти видаляє надлишкову інформацію з потоку вхідних значень, відображаючи дані в потік цілих чисел з однаковою бітовою довжиною, що полегшує стиснення. Фаза кодування вхідного потоку, згенерованого фазою карти, стискає його в більш конденсоване представлення. Карта має кілька варіантів виконання операції. Вона не робить ніяких перетворень. Обгрунтуванням є те, що дані всередині однієї підмножини напевно мають деяку схожість, тому повинні бути вже легко стиснені. Якщо розглядати їх як цілі числа знакової величини, то числа з плаваючою крапкою зберігають лексичний порядок своїх двійкових представлень. Тому, якщо двійкові представлення мають загальний префікс, то всі значення всередині діапазону мають однаковий префікс. Довжина загальної приставки визначається діапазонами плаваючих чисел між межами. Видалення цього префікса перетворює значення числа з плаваючою крапкою на менші цілі числа.

Розглянемо проблеми з непідписаним перевертанням. Проблема видалення префіксів полягає в тому, що числа з плаваючою комою представлені в знаковій величині. Тому, хоча значення мають лише невелику різницю, їх двійкові представлення не мають спільних префіксів бітів. Непідписані перевертання вирішують цю проблему шляхом зіставлення всіх чисел з плаваючою крапкою в суміжному діапазоні цілих чисел без знаку відповідно до їх значення. Це можна зробити, перевернувши біти знаків позитивних чисел і всі біти від'ємних чисел. Після віддзеркалення залишок обчислюється простим відніманням відображеної нижньої межі від перевернутого значення. Кодування після фази зіставлення використовує кодувальник для стиснення зіставлених даних у більш компактну форму. На рис. 1 зображено схему об'єднання індексованих масивів.

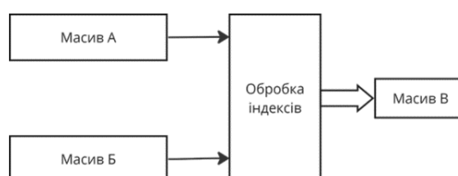

Рис. 1. Схема об'єднання індексованих масивів

Розглянемо чотири різні методи. Перший метод використовує алгоритм для стиснення зіставлених даних. Якщо зіставлені цілі числа завжди додатні, то він викликається безпосередньо. Якщо дані підписані перед компресором, то застосовується зигзагоподібне кодування, так що залишки з меншими абсолютними значеннями все ще зберігаються з меншою кількістю бітів. Другий метод спочатку обчислює різницю між зіставленими цілими числами. Потім він застосовує зигзагоподібне кодування до даних і використовує алгоритм для стиснення результатів. Третій метод використовує алгоритм стиснення плаваючих чисел для безпосереднього стиснення зіставлених даних. Четвертий метод є вдосконаленою версією другого алгоритму. Він спрямований на подолання двох недоліків другого методу, використовуючи його для стиснення обмежених значень. По-перше, при генерації хеш-значення метод витягує фіксовану кількість найбільш значущих бітів зі стисненого значення. Оскільки зіставлене ціле число, ймовірно, містить нуль бітів у більших бітах. Це призводить до меншої кількості інформації про використовуване значення. По-друге, алгоритм другого методу використовує модифікований формат варіантів байтів для стиснення різниці між фактичним значенням і

прогнозованим значенням. Цей формат може зберігати непотрібні нульові біти, а також вимагає три біти для кодування довжини закодованого числа. Кодувальник використовує ті ж предикати на основі хешу, що і компресор. Однак він налаштовує правий зсув бітів параметра двох предикатів відповідно до загальних префіксальних бітів верхньої та нижньої межі, так що біти вилучення не завжди включають деякі нульові біти. Згенерований залишок, також, стискається за допомогою алгоритму для кращого ступеня стиснення. Бітовий вектор селектора відстежує, який предикат використовується і передує стислому залишку. Всі методи відображення можуть бути векторизовані за допомогою інструкцій без умовних стрибків, а алгоритми стиснення оптимізовані для виконання. Поєднання різних методів відображення та кодування дає ряд можливих залишкових компресорів, але не всі комбінації мають сенс. Метод відображення може працювати тільки з кодувальником, який може ефективно стискати плаваючі дані. Тому, розглядаємо лише такі комбінації: видалення префікса, префікс видалення.

Розглянемо обробку запитів. Набір даних може бути як звичайним набором даних без індексу, так і індексованим. Обидва типи наборів даних надають API метод доступу згідно фрагментів для клієнтських програм. Система, також, надає набір складених операторів на основі витягування для користувачів, щоб легко отримувати доступ або обробляти ці набори даних, реалізовані за допомогою того ж API на основі фрагментів. Спочатку оператор зчитує набір даних, звичайний або індексований, зі сховища. Система, також, має можливість зчитування даних у зовнішніх форматах безпосередньо у вигляді звичайного набору даних через. Вибір на основі значень або розмірів може бути виконаний на сканованому наборі даних, перш ніж буде додатково оброблений батьківським оператором або клієнтською програмою. Індексовані та звичайні набори даних можуть бути перетворені один в одного за допомогою оператора, якщо батьківський оператор не підтримує інший тип набору даних.

Розглянемо метод доступу на основі фрагментів, наданий системою. Клієнт переглядає фрагменти набору даних і зчитує дані всередині фрагмента за допомогою чотирьох основних API для перегляду фрагментів у наборі даних. А, також, додаткові методи для запиту форми або типу фрагмента або бітів, які він містить. Більшість методів мають свою звичайну семантику. Однак, якщо тільки після отримання фрагмента користувач використовує ітератор фрагмента для навігації по його вмісту. Повторний виклик наступного *методу* переходить до наступної підмножини в множині. Для кожної підмножини значення та позиції елементів повертаються у менших блоках фіксованого розміру, які називаються *сегментами*, у тому ж порядку, що й звичайний набір даних. Порівняно з розпакуванням всієї підмножини одночасно, це дозволяє уникнути доступу до непотрібних елементів, а також покращує локальність кешу. Тому, покращує продуктивність сканування. Виклик наступного методу переводить систему до наступного сегмента та розпаковує її. Потім доступ до значень елементів у підмножині можна отримати за допомогою методу. Функція повертає інформацію про позицію, якщо її завантажено. Функція повертає лише непусті елементи. Порожні значення природно ігноруються через представлення сховища. Розглянемо, як метод доступу, може бути використаний для реалізації загальних операцій обробки масивів. Оператор вибирає значення в певному діапазоні в наборі даних. Спочатку він визначає набір бітів, який перетинається із запитаним діапазоном, і перебирає всі фрагменти, які містять ці біти. Для бітів у діапазоні запитів повністю розпакується та повертається весь біт. В іншому випадку він перевіряє елементи та повертає елементи в діапазоні запитів. Оператор фільтра завантажує позиційний індекс тільки в тому випадку, якщо батьківський оператор вимагає збереження вводу-виводу. Оператор повертає гіперпрямокутну розмірну область масиву. Спочатку він знаходить і повторює всі множини, що містять зацікавлену область. Якщо весь поділ покритий запитуваною областю, він повертає поділ безпосередньо. В іншому випадку він переглядає зміщення та повертає лише елементи, які знаходяться всередині області запиту. Оскільки перетворення зсувів на координати, а потім перевірка наявності умови запиту є обчислювально вартісним, оператор підмножини обчислює діапазони зсувів, запитаних за допомогою умови запиту, і перевіряє, чи збігаються розпаковані діапазони обчисленого діапазону зміщення. Оператор підмножини завжди завантажує інформацію про позицію, зчитуючи множини зі своїх дочірніх елементів. Оператор повертає значення агрегатів всіх елементів вхідного масиву. Він просто перебирає всі непорожні елементи і виконує необхідну агрегацію. При виконанні позиційно-залежних операцій над індексованим набором даних, таких як множення матриць або згортка, може бути бажано перетворити набір даних на звичайний поділ. Аналогічно, прості множині потрібно перетворити на індексовані множині під час завантаження даних. Система надає оператори для цієї потреби. Оператор приймає вхідний набір даних як вхід і перетворює його на звичайний набір даних з тією ж формою сегмента. Реалізація може виділяти простий поділ, перебирати всі індексовані множини, що перекриваються, один за одним, і де треба стиснути значення до їх правильних позицій. Однак це шкодить пам\яті, і додає значного тиску на систему виділення пам'яті і підкачки. Замість цього реалізація конструює і повертає простий поділ в одиниці *підмножини*, кожен з яких містить фіксовану кількість суміжних елементів в множині. Внутрішньо, метод працює подібно до об'єднання злиття: він читає наступні сегменти всіх перекритих бітів, і виконує операцію злиття, оскільки наступний сегмент запитується

батьківським оператором. Для обробки порожніх елементів він використовує бітовий вектор, щоб відстежити, які елементи не існують у всіх бітах, і повертає бітовий вектор разом з даними. Метод показує, що реалізація на основі ієрархії покращує вбудовану реалізацію. Оператор є оберненою операцією. На додаток до вхідного набору даних, йому також потрібно кількість бітів, діапазони кожного біту, а також кількість бітів у кожному фрагменті як параметри. Для кожної простої множини оператор використовує двійковий пошук для розміщення його значень у правильній підмножині, і комбінує біти для складання індексованих множин.

Порівняння результатів експериментів із запропонованою архітектурою сховища масивів з компактним інтегрованим індексом

Порівняємо із запитом даних приблизно на окремому растровому індексі, які компроміси здійснено з точки зору точності та часу запиту. Також, наявність можливості при необхідності ефективно перетворити відфільтрований результат індексованого набору даних на просту репрезентацію. Розглянемо параметри поділу, які підходять для індексованого набору даних та оцінюємо реалізацію на платформі. Дані зберігаються на локальному жорсткому диску. Система не реалізує саме управління кешем і спирається на кеш на рівні ОС і сховища. Очищаємо кеш до того, як всі експерименти були виміряні, час виконання. Експериментально порівнюється продуктивність тих самих запитів на індексованому масиві з виконанням звичайних наборів даних. Не існує механізму зберігання масивів з доступними індексами вартості. Щоб краще проілюструвати продуктивність системи, також, порівняємо її з популярною базою даних масивів, чий механізм зберігання даних показав подібну продуктивність вводу-виводу порівняно з популярними науковими форматами файлів. Як синтетичні, так і реальні набори даних використовуються для оцінки продуктивності запитів системи. Синтезований набір даних, який використовували, має рівномірний розподіл, і є двовимірним набором даних подвійних чисел з рівномірним розподілом на його домені значень. Два реальних набори даних взято. Перший набір містить ретроспективну та перспективну проекцію з кількома моделями. Перший набір даних, містить використання моделі. Набір даних можна розглядати як однопоточний масив з нестисненим розміром. Другий набір даних, що містить прогнозовану кількість значень, є однопоточним масивом з нестисненим розміром. Встановлюємо розмір множини таким чином, щоб кожен сегмент містив приблизно біля пів гігабайта даних. Якщо не буде іншого, то використовуємо біти еквівалентної ширини для індексованих наборів даних та автономних растрових індексів, а також зберігаємо біти в індексованому фрагменті. Вибір параметрів поділу ґрунтується на експериментах.

Вибираємо агрегацію з умовами вибору діапазону, оскільки це один з найпоширеніших запитів в аналітиці, і його ефективність є репрезентативною для різних запитів, що виконують індексоване сканування. Для перевірки продуктивності як вибору на основі значень, так і вибору на основі розмірів, використовуємо кілька умов вибору. Для запитів на основі значень продуктивність запиту може бути пов'язана як з тим, наскільки великий домен запиту як частка домену значень, так і з кількістю елементів, які запитуються. Невеликий обсяг пам'яті скорочує час вводу-виводу і часто призводить до меншого загального часу відповіді на запит, якщо введення-виведення є вузьким місцем досліджень. Досліджуємо зберігання залишкових бітів методів стиснення. Також, оцінюємо використання растрового стиснення або стиснення перевернутого списку щодо його більшої ефективності для зберігання позиційного індексу. Розмір стислих *залишкових* даних та *позиційний індекс* задамо у відсотках від вихідного набору даних. З точки зору залишкового стиснення, немає явного переможця. Вони є двома методами з найкращими коефіцієнтами стиснення. Хороша перфорація цих методів вказує на важливість для методу залишкового стиснення ефективно кодувати різницю в залишках. В цілому, перший метод є найкращим вибором. Навіть, коли він не досягає найкращого ступеня стиснення. Різниця зазвичай знаходиться в межах одного або двох відсотків від вихідного розміру даних. Оскільки перший метод захоплює більше інформації та кодує залишок більш ефективно, він також покращує ступінь стиснення. Метод відображення є більш ефективним, і це залежить від використовуваного методу кодування: непідписане перевертання краще працює з методом, тоді як *видалення префіксів* краще працює з кодувальником на певних наборах даних. Що стосується стиснення індексів, то інвертований список працює значно краще, ніж растровий малюнок в цілому, за винятком наборів даних з висококонцентрованими значеннями. Це пов'язано з тим, що растрові індекси, як правило, оптимізовані для швидких операцій перетину та об'єднання, а не для оптимізованого розміру сховища. Оскільки зіставлення збереженого зміщення з його залишковим значенням, а використання бітового зображення як індексу, також, потребує додаткового відстеження, то перевернутий список, як правило, більше підходить для зберігання позиційного індексу.

Згідно цих спостережень стиснули залишкові дані за допомогою першого методу, а позиційний індекс за допомогою другого методу в наступних експериментах. Це показує комбінований обсяг зберігання проіндексованого набору даних. Навіть з додатковим індексом все одно досягаємо кращого обсягу пам'яті порівняно з оригінальним набором даних, що ілюструє ефективність схеми індексованого зберігання. Оскільки набори даних зазвичай оновлюються нечасто, час створення індексу є одноразовою вартістю, а

ISSN 2219-9365

не головною турботою. Отже, до продуктивності індексного сканування системи розроблено відповідні методи. А виконання запиту на суму фільтра для індексованих наборів даних з виконанням тієї ж операції на звичайних наборах даних та оцінкою запиту. Продуктивність запиту фільтр-сума на однорідному наборі даних при зміні селективності від 0,7% до 100% є достатньою. Час відповіді на запит фільтрації індексованого набору даних пропорційний вибірковості, за винятком випадків, коли вибірковість нижча за 9%. Це пов'язано з тим, що механізм зберігання завантажує весь поділ, навіть якщо в ньому є доступ лише до однієї підмножини. Це вказує на те, що дисковий ввід/вивід завантаження даних, а не вартість розпакування, є фактичним вузьким місцем. Індексоване сканування також показує значну перевагу перед повним скануванням на звичайному наборі даних, час відгуку якого не змінюється з вибірковістю. При виборі 15% елементів операція фільтра на індексованому наборі даних дорівнює ∼ 11, що у 5 разів швидше, ніж на звичайному наборі даних.

Таким чином, проєктована система – система зберігання масивів з інтегрованою підтримкою індексу вартості. Система реорганізовує елементи в ряд користувацьких бітів і ефективно кодує індекси згідно бітів і відповідних їм значень, генеруючи індексоване представлення масиву, яке додає мало додаткових витрат на зберігання.

Висновки з даного дослідження і перспективи подальших розвідок у даному напрямі

Розроблена архітектура системи зберігання масивів з інтегрованою підтримкою індексу вартості. Завдяки їй досягається реорганізація елементів в ряд користувацьких бітів і ефективне кодування індексів згідно бітів і відповідних їм значень. При цьому генерується індексоване представлення масиву, яке додає мало додаткових витрат на зберігання.

Напрямами подальших досліджень є удосконалення архітектури системи зберігання масивів з інтегрованою підтримкою індексу вартості в частині оптимізації зберігання індексів.

Література

1. Dan A Alcantara, Vasily Volkov, Shubhabrata Sengupta, Michael Mitzenmacher, John D Owens, and Nina Amenta. Building an efficient hash table on the gpu. In *GPU Computing Gems Jade Edition*, pages 39–53. Elsevier, 2012.

2. Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkatara-man, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th {USENIX} Symposium onNetworked Systems Design and Implementation ({NSDI} 17)*, pages 469–482, 2017.

3. Witold Andrzejewski and Robert Wrembel. GPU-WAH: Applying GPUs to compressing bitmap indexes with word aligned hybrid. In *Lecture Notes in Com-puter Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6262 LNCS, pages 315–329. Springer, Berlin, Heidelberg, 2010.

4. Vo Ngoc Anh and Alistair Moffat. Index compression using 64-bit words. *Soft- ware - Practice and Experience*, 2010.

5. Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. Socrates: The new SQL server in the cloud. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1743–1756, 2019.

6. Lars Arge, Octavian Procopiuc, Sridhar Ramaswamy, Torsten Suel, and Jef- frey Scott Vitter. Scalable sweeping-based spatial join. In *VLDB*, volume 98, pages 570–581. Citeseer, 1998.

7. Manos Athanassoulis et al. Upbit: Scalable in-memory updatable bitmap in- dexing. In *SIGMOD '16*, 2016.

8. Utkarsh Ayachit, Andrew Bauer, Earl PN Duque, Greg Eisenhauer, Nicola Ferrier, Junmin Gu, Kenneth E Jansen, Burlen Loring, Zarija Lukic, Suresh Menon, et al. Performance analysis, design considerations, and applications of extreme-scale in situ infrastructures. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 921–932. IEEE, 2016.

9. Utkarsh Ayachit, Andrew Bauer, Berk Geveci, Patrick O'Leary, Kenneth More- land, Nathan Fabian, and Jeffrey Mauldin. ParaView catalyst: Enabling in situ data analysis and visualization. In *Proceedings of ISAV 2015: 1st International*

Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization, Held in conjunction with SC 2015: The International Conference for High Performance Computing, Networking, Storage and Analysis, pages 25–29, 2015.

10. Brian Babcock, Surajit Chaudhuri, and Gautam Das. Dynamic Sample Selec- tion for Approximate Query Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 539–550, 2003.

11. Alex R Van Ballegooij. RAM: A Multidimensional Array DBMS. *EDBT'04 Proceedings of the 2004 international conference on Current Trends in DatabaseTechnology*, pages 154–165, 2004.

12. C. Bauer, H. Abbasi, J. Ahrens, H. Childs, B. Geveci, S. Klasky, K. More- land, P. O'Leary, V. Vishwanath, B. Whitlock, and E. W. Bethel. In Situ Methods, Infrastructures, and Applications on High Performance Computing Platforms. In *Computer Graphics Forum*, volume 35, pages 577–597. Wiley Online Library, 2016.

13. Tekin Bicer and Gagan Agrawal. A Compression Framework for Multidimensional Scientific Datasets. In *IPDPS Workshop*, 2013.

14. Tekin Bicer, Jian Yin, and Gagan Agrawal. Improving I/O Throughput of Scientific Applications using Transparent Parallel Compression. In *CCGrid'14*, 2014.

15. Robert Bird, Patrick Killian, and Brian Albright. VPIC on GPU. *Bulletin of the American Physical Society*, 64, 2019.

References

1. Dan A Alcantara, Vasily Volkov, Shubhabrata Sengupta, Michael Mitzenmacher, John D Owens, and Nina Amenta. Building an efficient hash tableon the gpu. In *GPU Computing Gems Jade Edition*, pages 39–53. Elsevier, 2012.

2. Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkatara- man, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th {USENIX} Symposium onNetworked Systems Design and Implementation ({NSDI} 17)*, pages 469–482, 2017.

3. Witold Andrzejewski and Robert Wrembel. GPU-WAH: Applying GPUs to compressing bitmap indexes with word aligned hybrid. In *Lecture Notes in Com- puter Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6262 LNCS, pages 315–329. Springer,Berlin, Heidelberg, 2010.

4. Vo Ngoc Anh and Alistair Moffat. Index compression using 64-bit words. *Soft- ware - Practice and Experience*, 2010.

5. Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. Socrates: The new SQL server in the cloud. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1743–1756, 2019.

6. Lars Arge, Octavian Procopiuc, Sridhar Ramaswamy, Torsten Suel, and Jef- frey Scott Vitter. Scalable sweeping-based spatial join. In *VLDB*, volume 98, pages 570–581. Citeseer, 1998.

7. Manos Athanassoulis et al. Upbit: Scalable in-memory updatable bitmap in- dexing. In *SIGMOD '16*, 2016.

8. Utkarsh Ayachit, Andrew Bauer, Earl PN Duque, Greg Eisenhauer, Nicola Ferrier, Junmin Gu, Kenneth E Jansen, Burlen Loring, Zarija Lukic, Suresh Menon, et al. Performance analysis, design considerations, and applications of extreme-scale in situ infrastructures. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 921–932. IEEE, 2016.

9. Utkarsh Ayachit, Andrew Bauer, Berk Geveci, Patrick O'Leary, Kenneth More- land, Nathan Fabian, and Jeffrey Mauldin. ParaView catalyst: Enabling in situ data analysis and visualization. In *Proceedings of ISAV 2015: 1st International Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization, Held in conjunction with SC 2015: The International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 25–29, 2015.

10. Brian Babcock, Surajit Chaudhuri, and Gautam Das. Dynamic Sample Selec- tion for Approximate Query Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 539–550, 2003.

11. Alex R Van Ballegooij. RAM: A Multidimensional Array DBMS. *EDBT'04 Proceedings of the 2004 international conference on Current Trends in DatabaseTechnology*, pages 154–165, 2004.

12. C. Bauer, H. Abbasi, J. Ahrens, H. Childs, B. Geveci, S. Klasky, K. More- land, P. O'Leary, V. Vishwanath, B. Whitlock, and E. W. Bethel. In Situ Methods, Infrastructures, and Applications on High Performance Computing Platforms. In *Computer Graphics Forum*, volume 35, pages 577–597. Wiley Online Library, 2016.

13. Tekin Bicer and Gagan Agrawal. A Compression Framework for Multidimensional Scientific Datasets. In *IPDPS Workshop*, 2013.

14. Tekin Bicer, Jian Yin, and Gagan Agrawal. Improving I/O Throughput of Scientific Applications using Transparent Parallel Compression. In *CCGrid'14*, 2014.

15. Robert Bird, Patrick Killian, and Brian Albright. VPIC on GPU. *Bulletin of the American Physical Society*, 64, 2019.