12-2023

Index Bucketing: A Novel Approach to Manipulating Data Structures

Jeffrey Myers
Western Kentucky University, jeffrey.myers648@topper.wku.edu

INDEX BUCKETING:
A NOVEL APPROACH TO MANIPULATING NESTED DATA STRUCTURES

A Thesis submitted in partial fulfillment
of the requirements for the degree
Master of Science

Department of Computer Science
Western Kentucky University
Bowling Green, Kentucky

by
Jeffrey Myers II
December, 2023

Title:INDEX BUCKETING:A NOVEL APPROACH TO MANIPULATING NESTED DATA STRUCTURES

Name:Jeffrey Myers II

Date Recommended 12/1/2023

DocuSigned by:
Yaser Mowafi
50B70A68856F4C6...
Chair

DocuSigned by:
Huanjing Wang
A611FA899FDF476...
Committee Member

DocuSigned by:
Zhonghang Xia
205D70FB5B2B4B5...
Committee Member

Committee Member

DocuSigned by:
Jennifer Hammonds
EBE3858E068F42D...
Interim Director of the Graduate School

ABSTRACT

INDEX BUCKETING:
A NOVEL APPROACH TO MANIPULATING NESTED DATA STRUCTURES

Handling nested data collections in large-scale distributed systems poses considerable challenges in query processing, often resulting in substantial costs and error susceptibility. While substantial efforts have been directed toward overcoming computation hurdles in querying vast data collections within relational databases, scant attention has been devoted to the manipulation and flattening procedures necessary for unnesting these data collections. Flattening operations, integral to unnesting, frequently yield copious duplicate data and entail a loss of information, devoid of mechanisms for reconstructing the original structure. These challenges exacerbate in scenarios involving skewed, nested data with irregular inner data collections. Processing such data demands an extravagant number of operations, leading to extensive data duplication and imposing challenges in ensuring balanced distribution across partitions. Consequently, these factors impede performance and scalability. This research introduces a pioneering approach that amalgamates upfront computations with data manipulation techniques, specifically focusing on flattening procedures. This methodology aims to mitigate the adverse implications of data duplication and information loss while effectively addressing both skewed and irregular nesting structures. The efficacy of the proposed approach is assessed through comprehensive evaluations conducted on prominent datasets such as SQuAD, QuAC, and NewsQA, comparing its performance against existing methods like Pandas and recursive, iterative flattening implementations. These evaluations serve as a critical yardstick for gauging the effectiveness and viability of this novel approach in real-world scenarios.

Keywords: Irregular Schema, Skewed Distribution, Information Loss, Duplication Explosion, Nested Data Structures

In deep gratitude, I dedicate this work to the King of my heart and my life, whose subtle but essential guidance and providence I am convinced shaped the journey that lead to my discoveries. I am deeply grateful for God's grace, love, and mercy. He is with me through every step, never leaving my side and carrying me through where I fail.

"So do not fear, for I am with you; do not be dismayed, for I am your God. I will strengthen you and help you; I will uphold you with my righteous right hand."
– *Isaiah 41:10*

ACKNOWLEDGMENTS

First and foremost, I'd like to thank my advisor and mentor Dr. Yaser Mowafi for his crucial support and guidance throughout my research journey and for his constant encouragement. I've learned a lot from you, and I could not have done it without you!

To Drs. Guangming Xing, Huanjing Wang, Zhonghang Xia, Michael Galloway, and Mustafa Atici, thank you all for what you have taught me during my time at Western Kentucky University. Your guidance and support have been invaluable, and it has been my pleasure to study under your tutelage.

I am eternally grateful to my family for their unwavering, unconditional support and love through the ups and downs. Thanks Dad for believing in me and showing me patience and love, even during my worst moments. Thanks Mom for the sacrificial love and support that you demonstrate every day by what you do. Thanks to Denise, my other Mom who I am so blessed to have in my life. You're always there for me when I need words of encouragement or someone who will endure listening to my ramblings. Another thanks to my brother, David. We may butt heads a lot of the time, but I wouldn't have it any other way. Of those I consider my best friends, you are the best of the best. I love you all so much!

Last but not least, I want to thank those at Glasgow Bible Church whose fellowship has been a beacon of light to me and whose faithfulness has given me strength to persevere in life and in faith. I am eternally grateful for your love and support. You will always be in my heart, no matter where life takes me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

Introduction

The widespread rise in big data analytics has spurred interest in query processing systems that allow for performing complex analytical tasks over distributed data structures of arbitrary data types including nested data collections. Implementations of language-integrated query systems are evidenced in large-scale distributed data processing platforms such as Spark [2], Flink [4], Hadoop [7], and Microsoft LINQ [13]. Despite their vaunted support of nested data, these systems provide no direct processing for nested data over different distributed collections, whose values may themselves be collections. For example, previous work has demonstrated that nested algebra's lack of compositional operators of relational operators makes it unfeasible to directly express arbitrarily nested iterations over nested collections [5]. Expressing such complex computations on more complex data like nested collections often has a great impact on the overall execution cost, caused by so-called impedance mismatch issues. A term that refers to the problems arising when a data model defined by the user in the general-purpose programming language has to be flattened to match the tabular data model of the domain-specific language [6, 17].

To avoid this penalty, declarative language-integrated querying approaches have been proposed for processing nested APIs to integrate relational database queries with programming languages. For example, several proposals for this work provide embedded data-intensive scalable computing procedures that target current data analysis languages' query nesting and normalization for several distributed processing frameworks [1, 9, 10, 12, 21]. Similar to this is the exploration in calculus rules for mapping both set and multiset collection types and vice-versa for normalizing queries whose normal forms can be translated to SQL [16]. The work offers examples that do not appear straightforward to translate to SQL. Building on this calculus, Apache Hive [8], Google's F1 [18], and Spanner [3] provide SQL-like high-level languages with extensions for querying nested structured data at scale. The motivation for this work is to transform nested queries into efficient forms using relational algebra or set-oriented operators [11, 15, 22, 24]. Work committed by A. Ulrich [23] offers a review on

query flattening and descriptions of query flattening in database theory. It allows programmers to perform complex data analysis tasks, such as PageRank and matrix factorization using data-intensive scalable computing, called DIQL, that can run on various Big Data platforms, i.e., Apache Spark, Apache Flink, and Twitter's Cascading/Scalding [12].

Motivation - Navigating the complexities of handling nested data collections poses considerable computational hurdles, exacerbated by the inherent generation of copious duplicated data and redundant computations during unnesting procedures. These challenges are amplified within skewed nested data structures characterized by irregular inner data collections, where the endeavor to enforce balance across partitions escalates runtime inefficiencies and scalability limitations, exacerbating disk spillage and load imbalance issues [20].

This thesis is partly inspired by the work on an implementation technique for nested multisets, known as Query Shredding, which converts a nested complex query to a fixed number of flat query results [5]. The correspondence between the queries' multisets is maintained through binding semantics of indexed results for each shredded query. A recent work proposes a framework that translates a program manipulating nested collections into a set of semantically equivalent shredded queries of a sequence of queries, where both inputs and outputs may be nested and efficiently evaluated [20]. The work proposes a framework that flattens nested data queries by necessarily avoiding the nested structure entirely and instead utilizes a series of preprocessing and post-processing algorithms referred to as: Query Shredding and Query Stitching. While this approach has exhibited effectiveness in addressing many of these issues, it necessitates computationally intensive mechanisms due to the inherent lack of support for nested data structures within the confines of traditional relational database environments.

Objective - This work introduces a novel approach to addressing aggregation and flattening stages in both recent relational database methodologies and native document-oriented database systems, notably focusing on NoSQL databases like MongoDB [14] that inherently

support nested data structures. Despite the native querying support in document-oriented databases, their deficiency in providing an efficient process for manipulating nested structures perpetuates the challenges encountered in relational databases.

The proposed approach presented in this research offers a comprehensive solution capable of mitigating irregular nesting, skewed distribution, information loss, duplication explosion, and data partitioning challenges. Its efficacy extends beyond the scope of traditional relational database queries, addressing a persistent challenge in handling nested data structures efficiently.

Background

Relational Data Tables - To elucidate the functionality of the Index Bucketing algorithm and underscore the challenges it seeks to address; a straightforward question-and-answer dataset is employed. This dataset intricately articulates the nuances of skewed distribution and irregular schema, encapsulating its structure within four relational database tables: Sources, Questions, Answers, and Plausible Answers. For the sake of clarity and brevity, the number of records within a table is denote as n.

id (i)	context (ctx)
INC	STR

Table 1: Sources (src)

id (j)	text (txt)	i
INC	STR	FK

Table 2: Questions (qst)

Table 1 comprises source records featuring *id* and *context* fields. The *id* field encompasses incremental integers $i = \{1, \ldots, n\}$, while *context* stores textual excerpts extracted from source documents, such as paragraphs or articles. Table 2 incorporates *id*, *text*, and i fields. The *id* field embodies incremental integers $j = \{1, \ldots, n\}$, housing textual representations of questions like "How much time did John need to finish his book?" The i field functions as a foreign key referencing records in Table 1.

id (k)	start (srt)	end	j
INC	INT	INT	FK

Table 3: Answers (ans)

id (l)	start (srt)	end	j
INC	INT	INT	FK

Table 4: Plausible Answers (pls)

Table 3 encompasses *id*, *start*, *end*, and j fields. The *id* field spans incremental integers $k = \{1, \ldots, n\}$, while *start* and *end* signify the index positions of answers within the context of the related resource. The j field acts as a foreign key referring to records in Table 2, establishing nested relationships and facilitating the illustration of balanced and skewed data distribution. Introducing Table 4 enriches the dataset by mirroring fields akin to those in Table 3, with $l = \{1, \ldots n\}$ representing its incremental *id*. This table accommodates plausible yet indeterminate answers to questions, acknowledging instances where a definitive answer might be unattainable.

These interconnected tables establish a nested relationship structure, delineating diverse data distribution patterns while exemplifying irregular schema through the inclusion of Table 4. This comprehensive dataset provides an adequate landscape for illustrating the Index Bucketing algorithm's process to grapple with the complexities inherent in managing skewed distributions and irregular schemas.

Data Tree Representations - To further visualize the nested data structure portrayed by the relational tables, tree structures can be used to represent them. Figure 1, Figure 2, and Figure 3 are delineated to exemplify balanced, irregular, and skewed structures, respectively. Figure 4 amalgamates properties from Figure 2 and Figure 3, showcasing a blend of skewed and irregular characteristics. Within these tree representations, it becomes evident that sources might lack associated questions, and questions might encompass answers, plausible answers, both, or neither. This variability extends to the varying counts of answers and plausible answers within each question, along with fluctuations in the number of questions within each source. Such variability typifies an irregular nested structure marked by skewed data distribution.

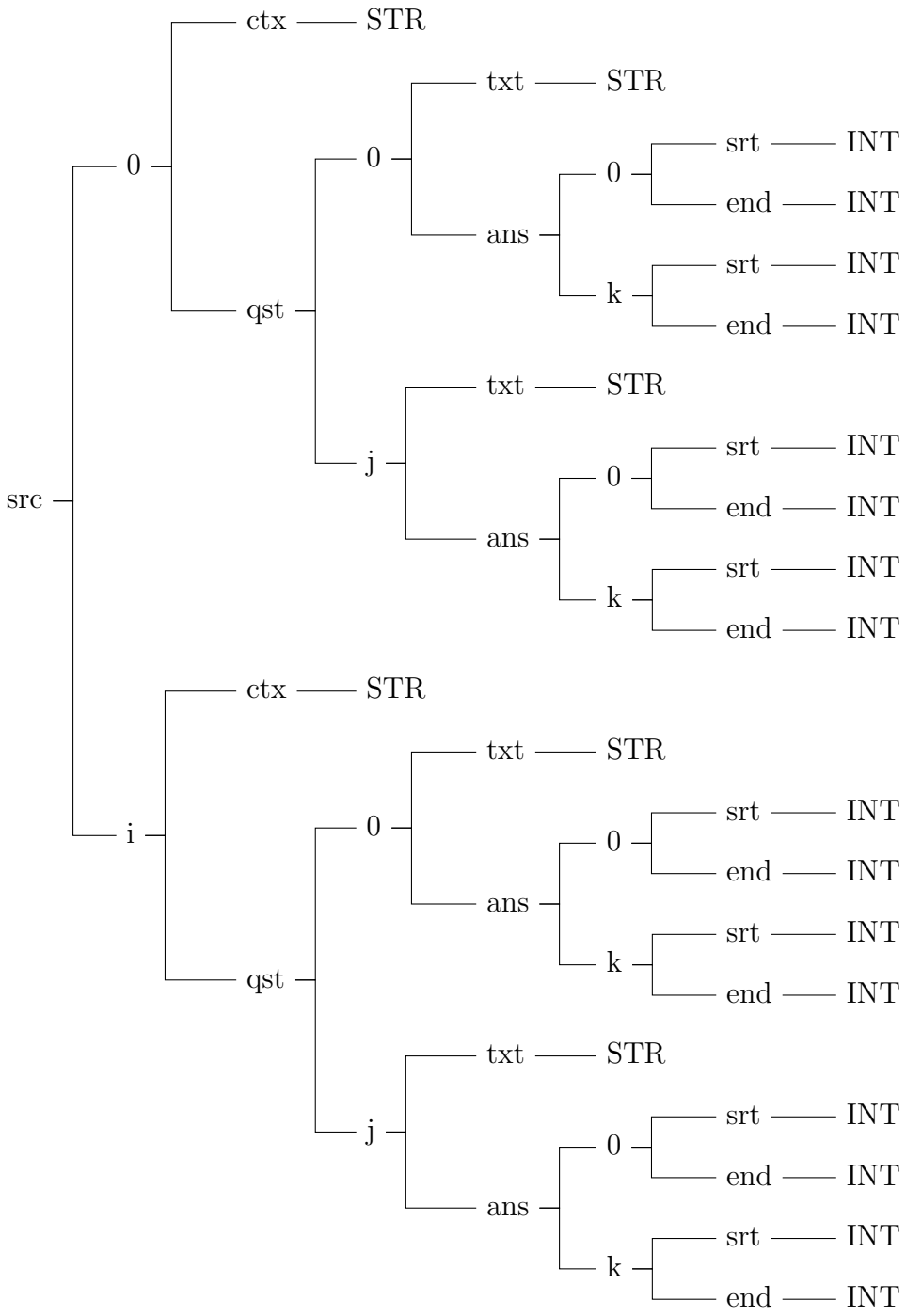

Figure 1: Balanced QAs Tree

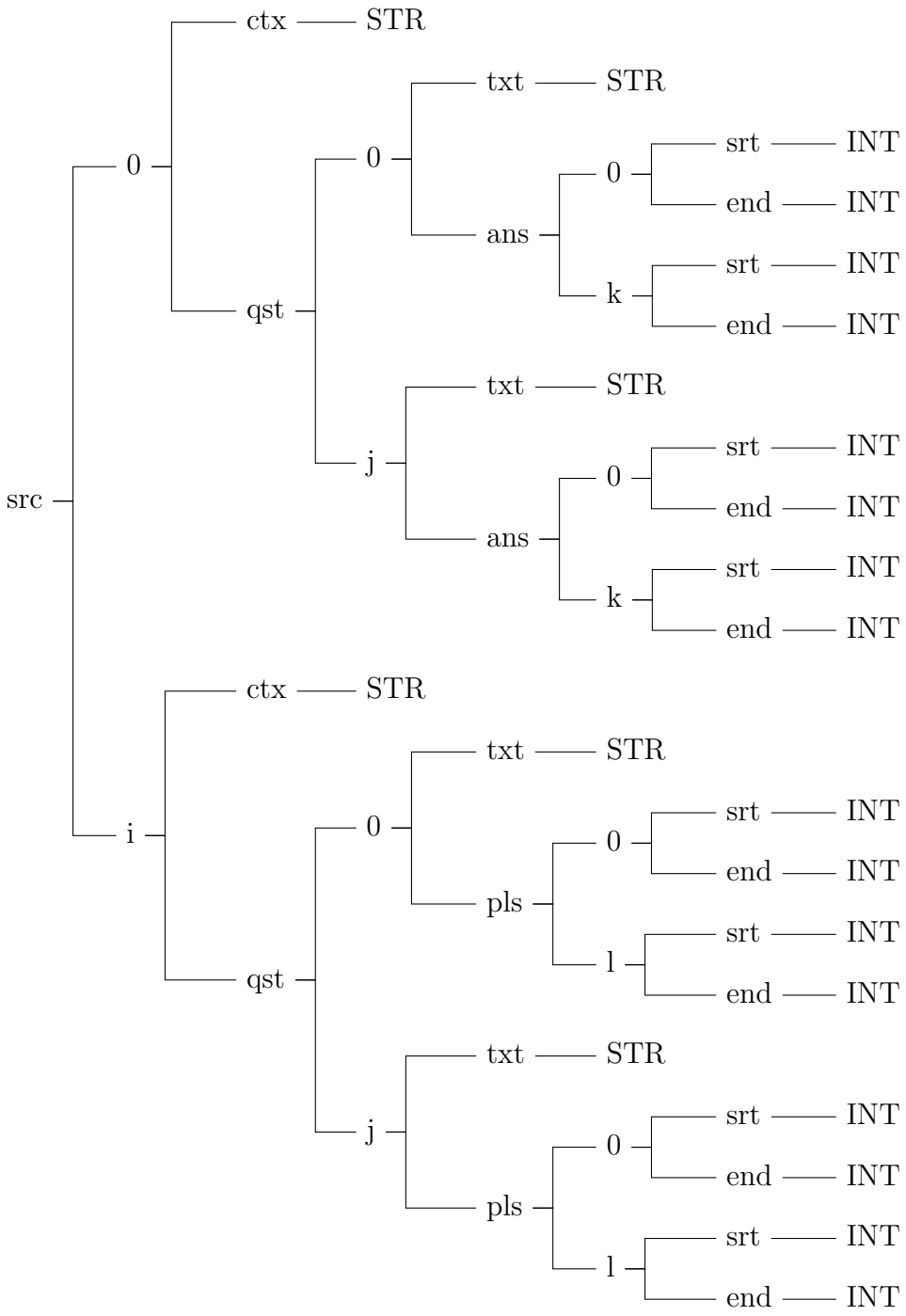

Figure 2: Irregular QAs Tree

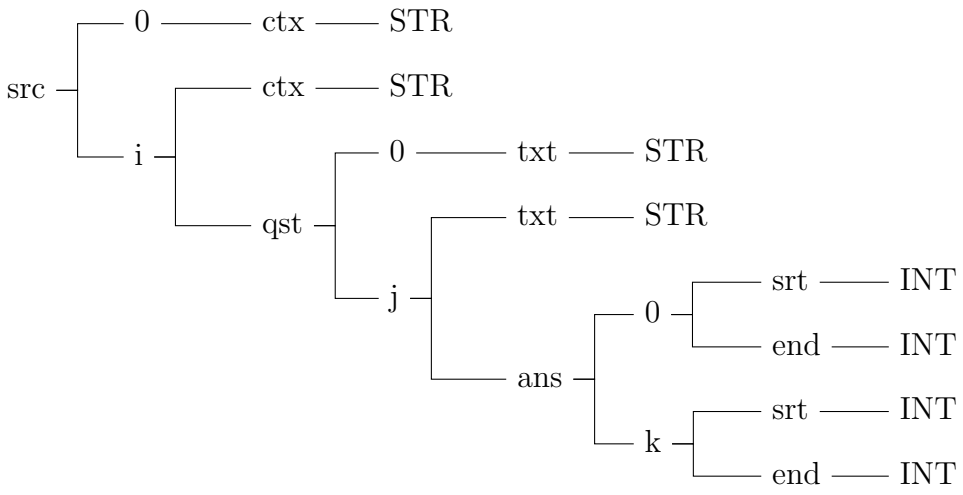

Figure 3: Skewed QAs Tree

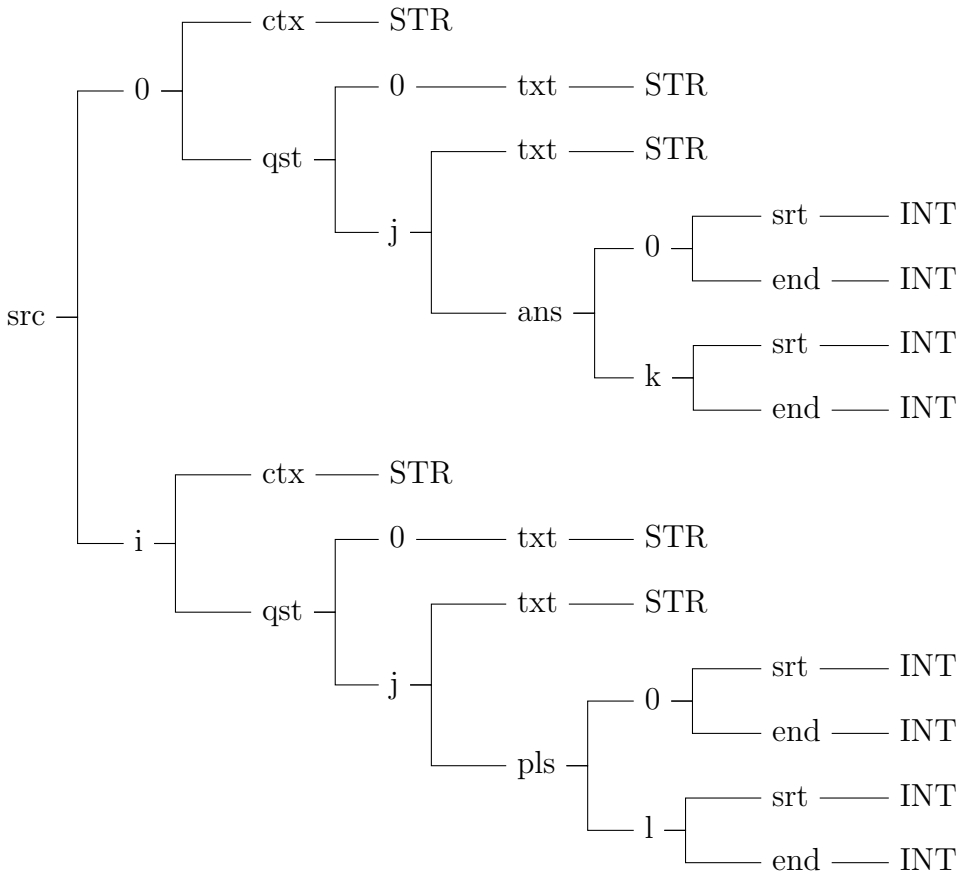

Figure 4: Skewed Irregular QAs Tree

This visualization not only illustrates the structural diversity but also hints at the applicability of the Index Bucketing algorithm within this context. The algorithm's nature, rooted in a tree-based framework, aligns with the illustrated tree structures and underscores the importance of indexes i, j, k, and l in comprehending and leveraging the Index Bucketing algorithm effectively. Considering nested data structures as akin to data trees lays the foundation for an algorithmic perspective essential in formulating efficient manipulation strategies for these complex structures.

Challenges

Addressing the complexities within nested data structures entails grappling with several formidable challenges, each posing unique obstacles that demand innovative solutions. Overcoming these multifaceted challenges necessitates innovative solutions, and the Index Bucketing algorithm emerges as a promising contender in mitigating these complexities. Its inherent efficiency in addressing issues of data duplication, skewed distributions, irregular schemas, and information loss within nested data structures positions it as a potential solution. As subsequent sections will delve into greater detail, Index Bucketing's capacity to optimize memory utilization, streamline data distribution, manage irregularities in schemas, and minimize information loss marks it as a valuable asset in navigating the intricacies of nested data manipulation.

Duplication Explosion - Also known as data avalanche or data storm, duplication explosion is a phenomenon characterized by an overwhelming proliferation of duplicated data during the flattening process [19]. As the term implies, this explosion results in an excessive replication of data, often leading to severe memory utilization issues and potential system failures, especially when handling extensive datasets. Current flattening solutions, primarily relying on recursion, fail to mitigate the adverse effects of this rampant data duplication.

Skewed Distribution - Another hurdle to overcome is skewed distribution, manifesting in nested data collections as unbalanced distributions of information [19]. When flattening such data, ensuring that each flattened instance contains all requisite keys introduces a problem akin to duplication explosion. However, in this case, missing keys necessitate filling with null values, requiring comprehensive parsing of the dataset to gather all keys. The challenge lies in devising an efficient algorithm to distribute these missing keys throughout the flattened data. Strategies may involve parsing before flattening, allowing simultaneous filling, or conducting a secondary traversal after flattening, although the former, while superior, presents implementation complexities.

Irregular Schema - Akin to skewed distribution, solving irregular schema involves filling missing keys throughout the dataset. However, it presents an even more intricate challenge. Here, disparate data collections within the dataset may contain entirely different keys at the same nesting level, significantly complicating parsing and filling algorithms.

Information Loss - The final challenge, information loss, poses a critical concern, describing the repercussions of flattening nested data structures. The flattened data loses crucial information required for reconstructing the original nested form. Without incorporating metadata into the flattened dataset, reconstructing the initial hierarchical structure becomes unfeasible. Reverting to the original data necessitates reloading the data file, which could be time-consuming and impractical for subsequent uses, especially with large datasets.

Index Bucketing Framework

The architecture of the Index Bucketing framework, rooted in a tree-based algorithmic approach, aligns seamlessly with the original nested data, preserving its inherent structure and circumventing potential information loss. This structural integrity stands as a robust solution within the algorithm, fundamentally resolving issues tied to data loss. Moreover, Index Bucketing strategically employs proactive processes, effectively offloading computa-

tional overheads that might impede performance during data flattening stages. During the initialization phase, comprehensive dataset analysis is executed, enabling the algorithm to adeptly navigate challenges stemming from skewed data distributions. This proficiency is achieved by aggregating index paths into an index bucket, a strategic mechanism facilitating efficient queries on nested data, ultimately producing flattened records. Addressing irregular schema, the initialization process includes constructing a flat template—a critical step ensuring every flattened record encompasses all absent keys filled with null values. This section delineates a concise implementation of the Index Bucketing framework, accentuating the algorithm's prowess in surmounting diverse challenges encountered in data management. The core attributes of this approach stand evident, exhibiting its adeptness in handling and mitigating a spectrum of complexities inherent in nested data structures.

Type Aliases - For the sake of simplicity, custom type aliases are introduced to represent types listed in their respective alias definitions. Additionally, the following types have been shortened for brevity:

String	Integer	Float	Complex	Boolean	Array	Tuple	Mapping
Str	Int	Flt	Cpx	Bln	Arr	Tup	Map

Algorithm 1 Type Aliases

alias Kdx ∈ {Str, Int}
alias Num ∈ {Int, Flt, Cpx}
alias Base ∈ {Num, Str, Bln, Null}
alias IPath ∈ {Arr[Int], Tup[Int]}
alias KPath ∈ {Arr[Str], Str}
alias IBckt ∈ {Set[IPath], Arr[IPath]}
alias KBckt ∈ {Set[KPath], Arr[KPath]}
alias Iter ∈ {Arr, Map, Set, Tup}
alias Val ∈ {Iter, Base}

Algorithm 2 Node Class

```
class Node
    function Node(kdx : Kdx, value : Val, level : Int, parent : Node)
        this.kdx : Kdx ← kdx
        this.value : Map[Node] or Base ← value
        this.level : Int ← level
        this.parent : Node ← parent
    function ipath → IPath
        return this.parent.ipath()
    function kpath → KPath
        return this.parent.kpath()
    function ibucket(depth : Int) → IBckt
        ibucket ← new Set[IPath]()
        for all child ∈ this.value do
            ibucket.update(child.ibucket(depth))
        return new Arr(ibucket).sort()
    function flatten(ipath : IPath) → Map
```

Base Node Definitions - As a foundational base class, the Node class serves as the common blueprint inherited by the Leaf, Branch, and Root classes within the Index Bucketing framework. The Node class lays out the essential structural elements shared across all inheriting classes:

- Node – This is the shared base constructor for all inheriting node classes and is responsible for setting the shared node attributes – *kdx*, *value*, *level*, and *parent*. The *kdx* attribute is a key or index value used for gathering index and key paths. The *value* attribute contains a collection of child Node types or serves as a Base value type for leaves. The *level* attribute is used to determine the depth of the node within the tree. The *parent* attribute is used to establish a link to the node's parent node.

- ibucket – By collecting a set of index paths, each aligning with the maximum *depth* of the nested data tree, this method is responsible for gathering the index bucket.

This standardized class structure established by the Node class ensures coherence and consistency in defining and organizing nodes across the Index Bucketing framework.

Algorithm 3 Leaf Classes

```
class Leaf inherits Node
    function Leaf(kdx : Kdx, value : Base, parent : Node)
        super(kdx, value, parent.level, parent)
    function ibucket(depth : Int) → IBckt
        ibucket ← new Set[IPath]()
        if this.level = depth then
            return ibucket.add(this.ipath())
        else return ibucket
    function flatten(ipath : IPath) → Map
        return new Map(this.kpath(), this.value)

class IndexedLeaf inherits Leaf // ILeaf or ILF
    function ipath → IPath
        ipath ← this.parent.ipath()
        return new Tup[Int](ipath.append(this.kdx))
    function kpath → KPath
        return new Str(".").join(this.parent.kpath())

class KeyedLeaf inherits Leaf // KLeaf or KLF
    function ipath → IPath
        return new Tup[Int](this.parent.ipath())
    function kpath → KPath
        kpath ← this.parent.kpath()
        return new Str(".").join(kpath.append(this.kdx))
```

Leaf Node Definitions - Within the framework, the Leaf class, along with its inheriting classes – IndexedLeaf and KeyedLeaf – fulfill the role of nodes encapsulating the terminus of nested data structures. These classes define essential functionalities pivotal to handling leaf nodes within the Index Bucketing framework:

- Leaf – Rather than directly receiving the *level* parameter argument, the Leaf constructor derives its *level* value from the *parent* node, ensuring hierarchical consistency within the tree structure.

- ibucket – This method accepts the maximum *depth* value of the tree as a parameter argument. It validates whether the *depth* value matches its *level*, subsequently

returning its index path enclosed in an index bucket set object if true; otherwise, an empty index bucket set object is returned. Employing a bottom-to-top algorithm, this method is invoked by non-leaf nodes to update and collate their child leaf node *value* fields into a set collection.

- FLATTEN – Disregarding the index path parameter argument, *ipath*, when invoked by the leaf node's corresponding parent, this method returns a new mapping of the leaf node's key path and *value*, adhering to a top-to-bottom calling sequence and resulting in a bottom-to-top return sequence.

- IPATH & KPATH – Defined in the INDEXEDLEAF and KEYEDLEAF classes which serve to differentiate leaves based on their indexing nature: indexed with integers or keyed with strings during tree initialization, these class methods manage bottom-to-top index paths or key paths by integrating the leaf node's *kdx* field along with its parent's index or key path, respectively. In cases where index paths are gathered, the leaf node converts arrays of index values into tuples of the same size.

By segregating leaves between indexed and keyed types during tree initialization, the classes circumvent the need for conditional evaluations. This strategic segregation bolsters performance and scalability, especially in managing larger datasets.

Algorithm 4 Branch Classes

```
class Indexed
    function ipath → IPath
        return this.parent.ipath().append(this.kdx)

class Keyed
    function kpath → KPath
        return this.parent.kpath().append(this.kdx)

class Branch inherits Node

class IndexingBranch inherits Branch
    function IndexingBranch(kdx : Int, value : Map[Node], parent : Node)
        super(kdx, value, parent.level + 1, parent)
    function flatten(ipath : IPath) → Map
        idx : Int ← ipath.at(this.level)
        if idx ∈ this.value.keys() then
            child : Node ← this.value.get(idx)
            return child.flatten(ipath)
        else return new Map()

class KeyingBranch inherits Branch
    function KeyingBranch(kdx : Str, value : Map[Node], parent : Node)
        super(kdx, value, parent.level, parent)
    function flatten(ipath : IPath) → Map
        flat ← new Map()
        for all child ∈ this.value do
            flat.update(child.flatten(ipath))
        return flat

class I2Branch inherits Indexed, IndexingBranch // I2B
class KIBranch inherits Keyed, IndexingBranch // KIB
class IKBranch inherits Indexed, KeyingBranch // IKB
class K2Branch inherits Keyed, KeyingBranch // K2B
```

Branch Node Definitions - The Branch class integrates into various specialized nodes, including I2Branch, KIBranch, IKBranch, and K2Branch which are defined by inheriting combinations of Indexed and Keyed classes with IndexingBranch and KeyingBranch classes.

- Indexed – The Indexed class encapsulates nodes indexed with integers, defining the ipath method to append the current node's index value to the parent's index path.
- Keyed – The Keyed class represents nodes keyed with strings, providing the kpath method to append the node's key value to the parent's key path.
- IndexingBranch – The IndexingBranch class inherits from Branch, designed for indexed branches. Its constructor sets attributes based on the provided values and parent node, and the flatten method retrieves the corresponding child node based on the index path.
- KeyingBranch – The KeyingBranch class, also extending Branch, targets keyed branches. Its constructor initializes attributes, and the flatten method iterates through child nodes, updating a map with their flattened results.
- I2Branch – I2Branch combines Indexed and IndexingBranch functionalities.
- KIBranch – KIBranch combines Keyed and IndexingBranch functionalities.
- IKBranch – IKBranch combines Indexed and KeyingBranch functionalities.
- K2Branch – K2Branch combines Keyed and KeyingBranch functionalities.

These specialized branch classes cater to different scenarios, providing distinct methods for handling various types of nested data collections. Each class offers unique functionalities for efficient execution, minimizing conditional evaluations during execution.

Algorithm 5 Root Classes

```
class ROOT inherits Node
    function ROOT(value : MAP[NODE], level : INT)
        SUPER(null, value, level, null)
    function IPATH → IPATH
        return new ARR()
    function KPATH → KPATH
        return new ARR()
    function FLATTEN(ibucket : IBCKT, template : MAP) → ARR[MAP]

class INDEXINGROOT inherits Root // IRoot or IRT
    function INDEXINGROOT(value : MAP[NODE])
        SUPER(value, 0)
    function FLATTEN(ibucket : IBCKT, template : MAP) → ARR[MAP]
        flats ← new ARR[MAP]()
        for all ipath ∈ ibucket do
            flat ← template.COPY()
            idx : INT ← ipath.AT(this.level)
            child : NODE ← this.value.GET(idx)
            flat.UPDATE(child.FLATTEN(ipath))
            flats.APPEND(flat)
        return flats

class KEYINGROOT inherits Root // KRoot or KRT
    function KEYINGROOT(value : MAP[NODE])
        SUPER(value, -1)
    function FLATTEN(ibucket : IBCKT, template : MAP) → ARR[MAP]
        flats ← new ARR[MAP]()
        for all ipath ∈ ibucket do
            flat ← template.COPY()
            for all child ∈ this.value do
                flat.UPDATE(child.FLATTEN(ipath))
            flats.APPEND(flat)
        return flats
```

ROOT NODE DEFINITIONS - The Root class, and its inheriting classes, marks the starting point of top-to-bottom processes and the conclusion of bottom-to-top processes within the Index Bucketing framework.

- ROOT – Inheriting from the Node class, the base Root class undergoes constructor modification, accepting solely *value* and *level* parameters. Root nodes lack *kdx* or *parent* attributes. Consequently, both the IPATH and KPATH methods return new empty arrays. Notably, the FLATTEN method's signature undergoes modification, now accepting the index bucket, *ibucket*, and flat *template* as parameters, and returning an array of flat mappings rather than a single mapping as seen in prior class definitions.

- INDEXINGROOT – This class inherits the base ROOT class, but its constructor configures the root node's level to 0 during instantiation, aligning its child node calling behavior with that of INDEXINGBRANCH nodes. Its FLATTEN method iterates over the index bucket, IBUCKET, dispatching each index path to the appropriate child nodes for further processing. An array of flat mappings, each of which are applied to a copy of the flat *template*, is gathered from the child nodes and is returned.

- KEYINGROOT – Also inheriting from the base ROOT class, the KEYINGROOT class sets its level to -1 within the constructor since its child calling behavior does not utilize the indexes from the index bucket. Its FLATTEN method operates in kind by passing index paths, IPATH, from the index bucket, IBUCKET, to its child nodes for further processing. Likewise, an array of flat mappings, each of which are applied to a copy of the flat *template*, is gathered from the child nodes and is returned.

By distinguishing between KEYINGROOT and INDEXINGROOT nodes, the tree's root node ensures that subsequent *level* attributes are set appropriately during initialization and the index bucket is distributed accordingly during execution.

Algorithm 6 Tree Class

```
class Tree
    function Tree(data : Iter)
        this.depth : Int ← 0
        this.kbucket : KBckt ← new Set()
        this.tree : Root ← this.root(data)
        this.ibucket : IBckt ← this.tree.ibucket(this.depth)
        this.template ← new Map()
        for all kpath ∈ this.kbucket do
            this.template.update(new Map(kpath, null))
    function flatten → Arr[Map]
        return this.tree.flatten(this.ibucket, this.template)
    function leaf(kdx : Kdx, data : Base, parent : Node) → Leaf
    function branch(kdx : Kdx, data : Iter, parent : Node) → Branch or Leaf
    function root(data : Iter) → Root
```

Tree Structure Definitions - The Tree class serves as the foundational structure to organize the nested dataset for the execution of the Index Bucketing algorithm.

- Tree – In the constructor, the initialization commences by setting the *depth* field to 0 and creating an empty set object for the key bucket, *kbucket*. These fields are then used to analyze the *data* parameter's nested structure while the tree itself is constructed and stored within the *tree* field which acts as a reference to the root node. Next, the algorithm gathers the index bucket, *ibucket*. Additionally, it constructs the *template* by iterating through the key bucket, compiling all key paths into a mapping with initial null values for each key path. This flat template formation streamlines the subsequent data organization process.

- flatten – To facilitate the flattening process, the Tree class defines its own flatten method. This method initiates the root node's flatten method, passing along the index bucket and flat template.

Additionally, the Tree class establishes three methods – leaf, branch, and root – to assemble the appropriate node types for building the tree structure.

Algorithm 7 Tree Leaf Method

```
function LEAF(kdx : KDX, data : BASE, parent : NODE) → LEAF
    leaf : LEAF ← null
    if TYPE(kdx) = INT then
        leaf ← new INDEXEDLEAF(kdx, data, parent)
    else leaf ← new KEYEDLEAF(kdx, data, parent)
    this.depth ← MAX(this.depth, leaf.level)
    this.kbucket.ADD(leaf.KPATH())
    return leaf
```

Unlike the other node methods defined in the TREE class, the LEAF method not only initializes the relevant LEAF class node, it also identifies the maximum depth of the tree. This DEPTH determination is crucial for collecting index paths into the index bucket. Moreover, the method aggregates key paths into the key bucket and, finally, returns the instantiated Leaf node.

Algorithm 8 Tree Branch Method

```
function BRANCH(kdx : KDX, data : ITER, parent : NODE) → BRANCH or LEAF
    if LENGTH(data) > 0 then
        branch : BRANCH ← null
        if TYPE(data) ≠ MAP then
            data ← ENUMERATE(data)
            if TYPE(kdx) = STR then
                branch ← new KIBRANCH(kdx, new MAP(), parent)
            else branch ← new I2BRANCH(kdx, new MAP(), parent)
        else if TYPE(kdx) = STR then
            branch ← new K2BRANCH(kdx, new MAP(), parent)
        else branch ← new IKBRANCH(kdx, new MAP(), parent)
        for all kdx, value ∈ data.ITEMS() do
            node : NODE ← null
            if TYPE(value) = ITER then
                node ← this.BRANCH(kdx, value, branch)
            else node ← this.LEAF(kdx, value, branch)
            branch.value.UPDATE(kdx, node)
        return branch
    else return this.LEAF(kdx, null, parent)
```

In the BRANCH method, if the passed *data* collection is not empty, this method initializes the appropriate BRANCH class node. Conversely, if the collection is empty, it delegates the parameter arguments to the LEAF method with null passed as the *data* parameter's argument. When the BRANCH method's *data* parameter isn't empty, it also iterates through the collection. Depending on the nested data type, it further directs nested information to either another BRANCH method call or a LEAF method call. Lastly, the constructed BRANCH node is returned.

Algorithm 9 Tree Root Method

```
function ROOT(data : ITER) → ROOT
    if LENGTH(data) > 0 then
        root : ROOT ← null
        if TYPE(data) ≠ MAP then
            data ← ENUMERATE(data)
            root ← new INDEXINGROOT(new MAP())
        else root ← new KEYINGROOT(new MAP())
        for all kdx, value ∈ data.ITEMS() do
            node : NODE ← null
            if TYPE(value) = ITER then
                node ← this.BRANCH(kdx, value, branch)
            else node ← this.LEAF(kdx, value, branch)
            root.value.UPDATE(kdx, node)
        return root
    else return null
```

Differing from the BRANCH method, the ROOT method returns null when the *data* parameter is an empty collection, indicating that no data is present. However, when the collection holds nested information, the ROOT method initializes the appropriate ROOT class node. It then systematically traverses through the nested data of the collection. Depending on the nested data type, it appropriately calls either the BRANCH method or the LEAF method. Finally, the method returns the constructed Root node, forming the basis of the tree structure.

Algorithm 10 Generator Implementation

```
class IndexingRoot inherits Root
    function flatten(ipath : IPath, template : Map) → Map
        idx : Int ← ipath.at(this.level)
        child : Node ← this.value.get(idx)
        flat ← template.copy()
        flat.update(child.flatten(ipath))
        return flat

class KeyingRoot inherits Root
    function flatten(ipath : IPath, template : Map) → Map
        flat ← template.copy()
        for all child ∈ this.value do
            flat.update(child.flatten(ipath))
        return flat

class Tree
    function Tree(kdx : Kdx, data : Iter)
        this.count : Int ← 0
        this.depth : Int ← 0
        this.kbucket : KBckt ← new Set()
        this.tree : Root ← this.root(kdx, data)
        this.ibucket : IBckt ← this.tree.ibucket(this.depth)
        this.template ← new Map()
        for all kpath ∈ this.kbucket do
            this.template.update(new Map(kpath, null))
    function flatten → Map
        if this.count ≥ length(this.ibucket) then
            this.count ← 0
            return null
        this.count ← this.count + 1
        ipath : IPath ← this.ibucket.at(this.count)
        return this.tree.flatten(ipath, this.template)
```

Up to now, the implementation used for this thesis has been defined and described. However, to demonstrate the implementation flexibility of the Index Bucketing algorithm, an alternative implementation is presented. Through subtle modifications to the Root and Tree class definitions, the framework transforms into a generator capable of delivering flattened data incrementally rather than in a single instance. Instead of the Root node

managing the index bucket within its FLATTEN method, this responsibility is shifted to the TREE class's FLATTEN method. Introducing a *count* field, initialized at 0, enables the tracking of index bucket progress. When *count* reaches the end of the index bucket, it is reset to 0, and null is returned to signal completion. This generator-style implementation offers a method to alleviate the adverse effects of duplication explosion which can otherwise overload memory usage. The adaptability of Index Bucketing as an algorithm allows for diverse implementations, offering various advantages to address challenges that stem from previous approaches.

Experimental Evaluation

To assess the efficacy of the Index Bucketing algorithm, evaluations delved into performance measurements across three prominent question-answering datasets: SQuAD, QuAC, and NewsQA. These datasets vary in file size: 44.3 MB, 73.4 MB, and 151 MB respectively. The Index Bucketing algorithm was juxtaposed against two alternative flattening implementations: one leveraging the Pandas Python package, and another employing a basic solution that combines recursive and iterative techniques. The Pandas Python package is used as a baseline for comparison, as it offers an explode function that can quickly flatten nested data collections. The basic implementation, on the other hand, serves to demonstrate the worst-case effects of each challenge previously described.

Evaluations spanned various subsets of each dataset incrementally from a Fibonacci-based sequence on the range $0.1\% \rightarrow 100\%$ to gauge scalability. Each subset underwent three evaluations, and the average runtimes across these executions were recorded to ensure more robust assessments. Each evaluation observed initialization, execution, and total runtimes.

The ensuing graphs are organized by implementation and dataset, plotting subset size, measured in bytes, against runtime, measured in seconds. These evaluations were conducted on an MSI GP63 Leopard 8RE laptop, with specifications of 32GB RAM and an Intel Core i7-8750H CPU clocking in at a base frequency of 2.20GHz, capable of reaching a maximum

turbo frequency of 4.10GHz.

It's important to note that these evaluations were performed without parallel processing, and a stringent maximum time limit of thirty minutes was set to avoid prolonged executions, triggering a timeout exception if exceeded. Framework and experimental implementation can be found in the Github project repository at:

`https://github.com/JeffMII/Index-Bucketing`

Evaluation Tables - The following tables detail the evaluation results. Subsequent subsections summarize this information in plotted graphs, and observations for each set of evaluations are discussed. Note that a missing table signifies that the implementation failed on that dataset, while missing rows signify that the implementation timed out for those subsets.

Ratio	Count	Initial Time	Execution Time	Total Time	Initial Size
0.001	1	0.0168962	0.0114848	0.0283811	205947
0.002	1	0.0513001	0.0071109	0.0584110	205947
0.003	1	0.0216508	0.0069941	0.0286450	205947
0.005	2	0.0304157	0.0135179	0.0439336	404577
0.008	3	0.0753669	0.0162261	0.0915931	523935
0.013	6	0.1111545	0.0271083	0.1382629	851257
0.021	10	0.1419780	0.0437985	0.1857765	1443481
0.034	16	0.2510539	0.0704269	0.3214808	2368409
0.055	26	0.3615115	0.0925745	0.4540860	3075819
0.089	42	0.4877568	0.1274244	0.6151813	4203649
0.144	68	0.7213556	0.1938976	0.9152533	6368153
0.233	111	1.1571026	0.3174184	1.4745210	10066704
0.377	179	1.7752043	0.4970038	2.2722082	16464241
0.61	290	2.9855251	0.8255536	3.8110788	27363292
0.987	470	5.0167932	1.4272351	6.4440283	45424399
1.0	477	5.1734913	1.5167224	6.6902138	46494152

Table 5: Index Bucketing on SQuAD

Ratio	Count	Initial Time	Execution Time	Total Time	Initial Size
0.001	12	0.0026879	0.0013852	0.0040731	69326
0.002	25	0.0072623	0.0030416	0.0103039	150958
0.003	37	0.0087714	0.0054370	0.0142084	221714
0.005	62	0.0151363	0.0072584	0.0223948	372253
0.008	100	0.0538734	0.0165209	0.0703943	586960
0.013	163	0.0697477	0.0174856	0.0872334	946716
0.021	263	0.0928620	0.0288163	0.1216783	1541099
0.034	427	0.1619864	0.0471399	0.2091264	2515197
0.055	691	0.2654181	0.0746883	0.3401064	4084462
0.089	1118	0.4096990	0.1256608	0.5353599	6623965
0.144	1809	0.7218658	0.2061604	0.9280263	10685287
0.233	2928	1.0858766	0.3170777	1.4029543	17187531
0.377	4737	1.7819539	0.5162077	2.2981617	27841109
0.61	7665	2.8399793	0.8583483	3.6983276	45035848
0.987	12403	4.9677818	1.7010544	6.6688363	75588570
1.0	12567	5.0548561	1.7414888	6.7963450	77044015

Table 6: Index Bucketing on QuAC

Ratio	Count	Initial Time	Execution Time	Total Time	Initial Size
0.001	12	0.0085266	0.0053094	0.0138360	73736
0.002	25	0.0143535	0.0088745	0.0232280	156085
0.003	38	0.0252807	0.0166043	0.0418851	264572
0.005	63	0.0410175	0.0275607	0.0685782	421730
0.008	101	0.1697030	0.0437680	0.2134710	682251
0.013	165	0.2187332	0.0718318	0.2905651	1150158
0.021	267	0.2858403	0.1145694	0.4004098	1873578
0.034	433	0.4969543	0.1957834	0.6927378	3058549
0.055	700	0.6991990	0.3014860	1.0006851	4888043
0.089	1134	1.2761940	0.4917707	1.7679648	7941941
0.144	1835	2.0246188	0.8012632	2.8258821	12875802
0.233	2969	3.1936919	1.3475715	4.5412634	20863684
0.377	4804	5.0125851	2.0572125	7.0697977	33613282
0.61	7773	7.9774706	3.2619555	11.2394261	54097739
0.987	12578	12.5392207	5.2448907	17.7841115	86775720
1.0	12744	12.7942890	5.1597726	17.9540617	87916829

Table 7: Index Bucketing on NewsQA

Ratio	Count	Initial Time	Execution Time	Total Time	Initial Size
0.001	1	0.0002762	0.0746655	0.0749418	205947
0.002	1	0.0002634	0.0760294	0.0762928	205947
0.003	1	0.0002606	0.0764116	0.0766722	205947
0.005	2	0.0002826	0.0932931	0.0935757	404577
0.008	3	0.0002847	0.1042973	0.1045820	523935
0.013	6	0.0003005	0.1646125	0.1649130	851257
0.021	10	0.0004242	0.2297497	0.2301740	1443481
0.034	16	0.0004159	0.3446137	0.3450296	2368409
0.055	26	0.0003610	0.4462409	0.4466020	3075819
0.089	42	0.0005614	0.5559562	0.5565177	4203649
0.144	68	0.0004827	0.8514183	0.8519010	6368153
0.233	111	0.0005594	1.2751993	1.2757587	10066704
0.377	179	0.0004956	2.0982393	2.0987350	16464241
0.61	290	0.0007402	3.5251046	3.5258449	27363292
0.987	470	0.0008452	6.2241778	6.2250231	45424399
1.0	477	0.0010821	6.4968896	6.4979718	46494152

Table 8: Pandas on SQuAD

Ratio	Count	Initial Time	Execution Time	Total Time	Initial Size
0.001	12	0.0003564	0.1230967	0.1234532	69326
0.002	25	0.0002728	0.1300340	0.1303068	150958
0.003	37	0.0003091	0.1374667	0.1377758	221714
0.005	62	0.0003536	0.1520365	0.1523902	372253
0.008	100	0.0004118	0.1716765	0.1720884	586960
0.013	163	0.0004774	0.2128723	0.2133498	946716
0.021	263	0.0006955	0.2583871	0.2590826	1541099
0.034	427	0.0009579	0.3451617	0.3461197	2515197
0.055	691	0.0013662	0.5085468	0.5099131	4084462
0.089	1118	0.0019558	0.7457041	0.7476599	6623965
0.144	1809	0.0019939	1.1232026	1.1251966	10685287
0.233	2928	0.0038892	1.7710431	1.7749323	17187531
0.377	4737	0.0048408	2.8533043	2.8581451	27841109
0.61	7665	0.0076718	4.6776418	4.6853137	45035848
0.987	12403	0.0131796	8.1195218	8.1327015	75588570
1.0	12567	0.0131604	9.0104400	9.0236004	77044015

Table 9: Pandas on QuAC

Ratio	Count	Initial Time	Execution Time	Total Time	Initial Size
0.001	1	0.0	0.2359724	0.2359724	205947
0.002	1	0.0	0.2413332	0.2413332	205947
0.003	1	0.0	0.2325857	0.2325857	205947
0.005	2	0.0	1.3330778	1.3330778	404577
0.008	3	0.0	2.6223285	2.6223285	523935
0.013	6	0.0	11.5684023	11.5684023	851257
0.021	10	0.0	50.8342733	50.8342733	1443481
0.034	16	0.0	176.8247058	176.8247058	2368409
0.055	26	0.0	335.1017868	335.1017868	3075819
0.089	42	0.0	680.1228720	680.1228720	4203649
0.144	68	0.0	1715.3059863	1715.3059863	6368153

Table 10: Basic on SQuAD

Ratio	Count	Initial Time	Execution Time	Total Time	Initial Size
0.001	12	0.0	0.0170995	0.0170995	69326
0.002	25	0.0	0.0430601	0.0430601	150958
0.003	37	0.0	0.0761557	0.0761557	221714
0.005	62	0.0	0.1495611	0.1495611	372253
0.008	100	0.0	0.2990935	0.2990935	586960
0.013	163	0.0	0.6933944	0.6933944	946716
0.021	263	0.0	4.8759892	4.8759892	1541099
0.034	427	0.0	21.5186237	21.5186237	2515197
0.055	691	0.0	83.0441016	83.0441016	4084462
0.089	1118	0.0	262.8868047	262.8868047	6623965
0.144	1809	0.0	768.5876779	768.5876779	10685287

Table 11: Basic on QuAC

Ratio	Count	Initial Time	Execution Time	Total Time	Initial Size
0.001	12	0.0	0.0435486	0.0435486	73736
0.002	25	0.0	0.1434285	0.1434285	156085
0.003	38	0.0	0.3722347	0.3722347	264572
0.005	63	0.0	1.3329352	1.3329352	421730
0.008	101	0.0	6.3334334	6.3334334	682251
0.013	165	0.0	32.0920742	32.0920742	1150158
0.021	267	0.0	109.3982449	109.3982449	1873578
0.034	433	0.0	351.9411959	351.9411959	3058549
0.055	700	0.0	1000.5315140	1000.5315140	4888043

Table 12: Basic on NewsQA

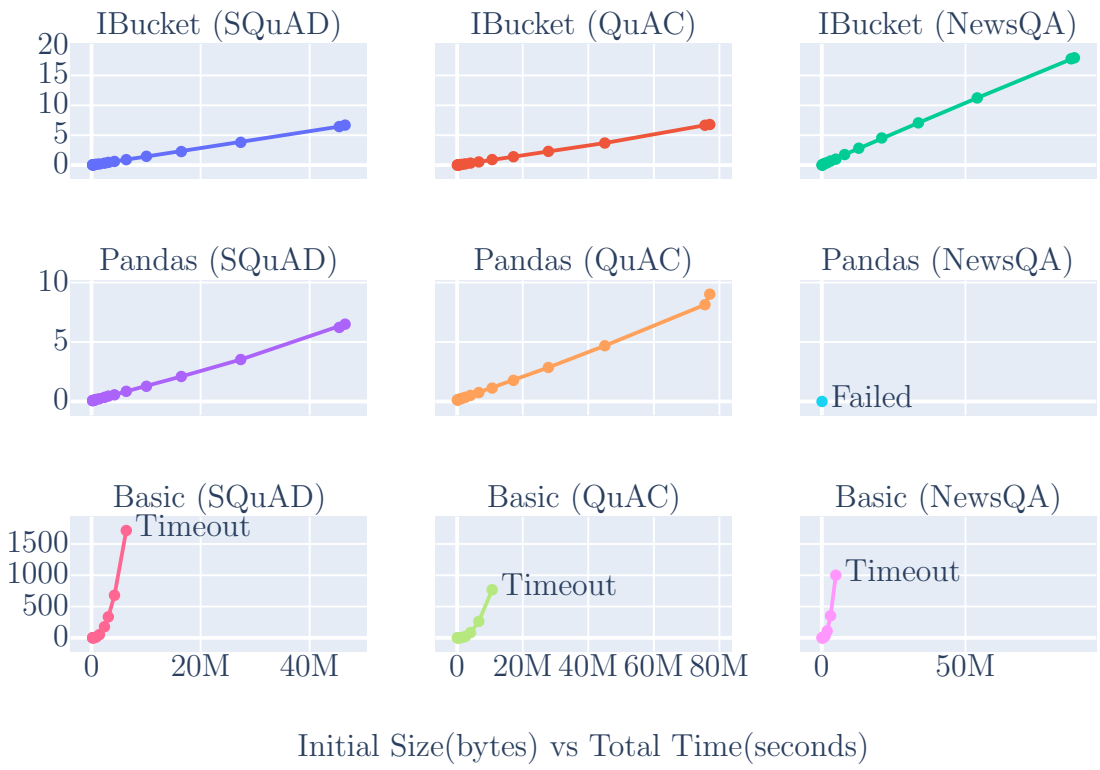

Figure 5: Total Time Evaluation Plots

Total Time - Before delving into the detailed examination of initialization and execution runtimes, an initial analysis of total runtimes across implementations provides valuable insights.

Notably depicted in Figure 5, the basic algorithm showcases an exponential growth pattern in total runtimes, vividly illustrating the cost escalations attributed to challenges that the Index Bucketing algorithm aims to address. A closer inspection reveals a near alignment between the runtimes of Index Bucketing and Pandas implementations. However, Index Bucketing consistently demonstrates superior performance, more notable the larger the dataset. Another important observation is that, during evaluations with NewsQA data, Pandas encounters failures due to duplicated data instances within the original dataset. While Pandas offers potential solutions to address these errors, implementing such remedies remains complex and non-trivial based on existing knowledge.

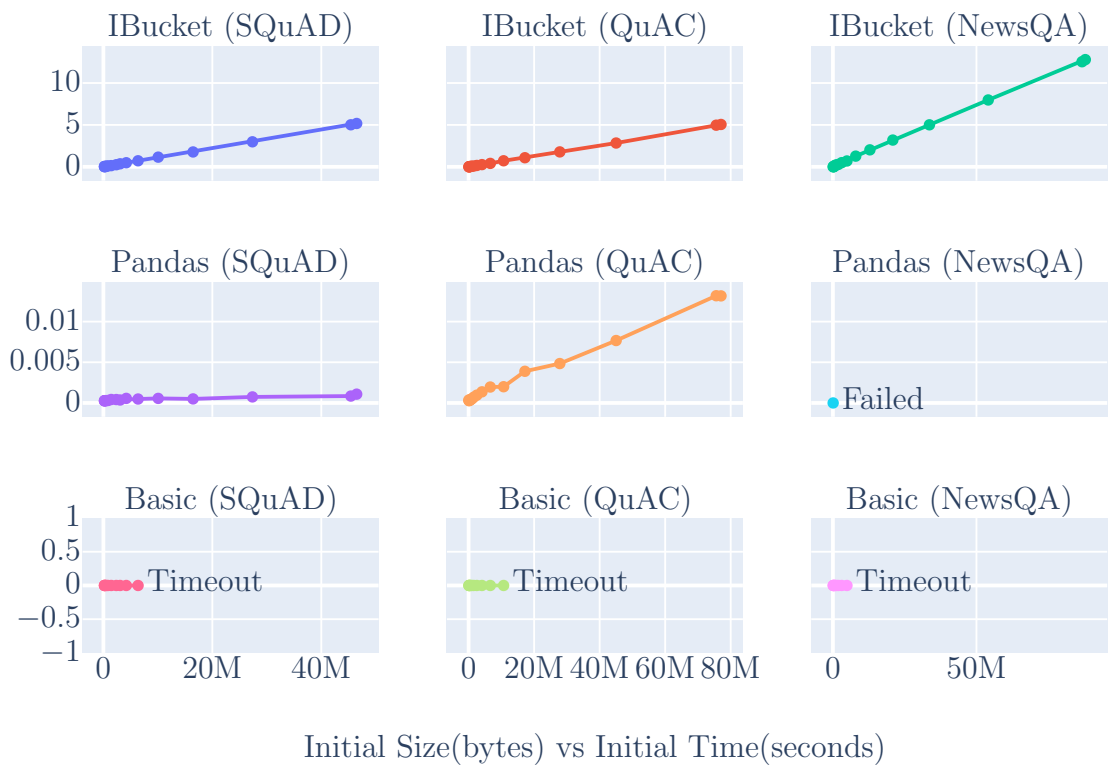

Figure 6: Initialization Time Evaluation Plots

Initialization Time - Following the dataset's loading into memory, both the Index Bucketing and Pandas implementations undergo specific initialization procedures. In contrast, the basic implementation, lacking a framework encapsulating the dataset, doesn't involve any observed initialization times, therefore they are recorded as 0. For Pandas, its very small initialization times are attributed to the creation of a DataFrame object where the nested data resides. As DataFrame objects don't directly interpret nested data collections, their initialization times are nearly 0, as shown in Table 10 and Table 11.

Conversely, Index Bucketing allocates significant time during initialization for comprehensive dataset analyses. Consequently, the initialization time for the Index Bucketing implementation significantly surpasses that of the other approaches. Despite this longer initialization period, the scalability remains linear. These observations highlight how the Index Bucketing framework strategically shifts a considerable workload to the initialization

process, optimizing subsequent execution times, as evident in the ensuing evaluation plots.

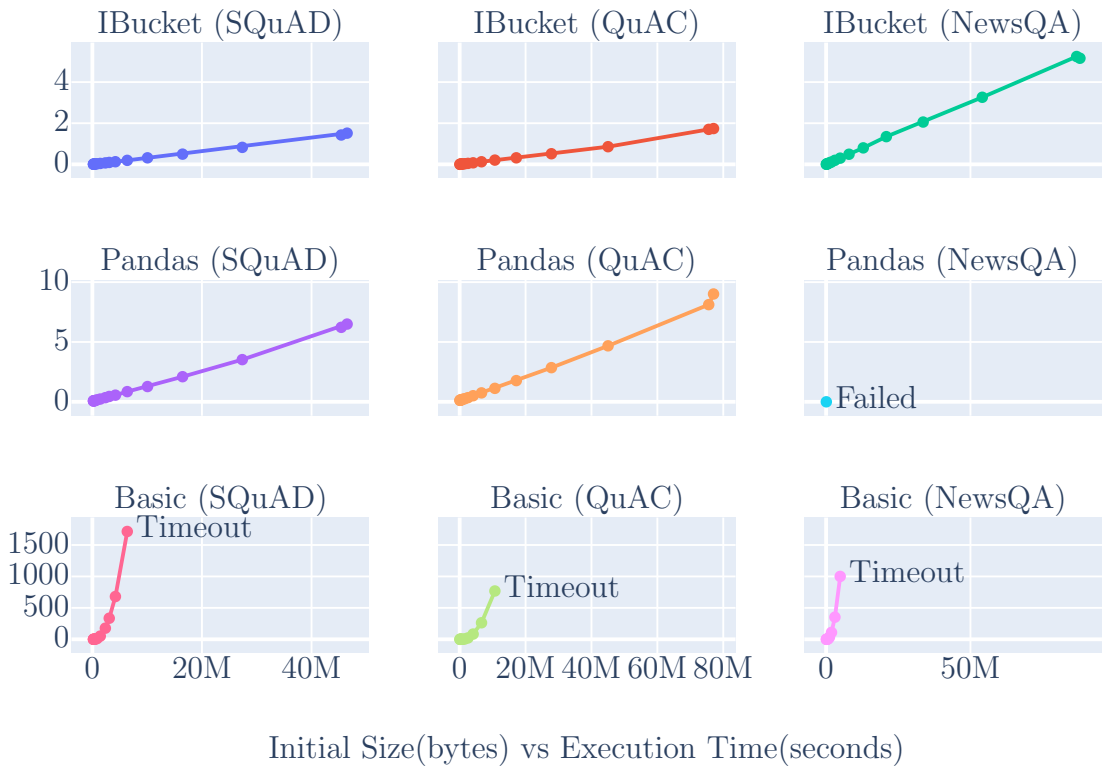

Figure 7: Execution Time Evaluation Plots

Execution Time - In observing the execution runtimes in Figure 7, it's apparent that Index Bucketing significantly surpasses the other implementations in performance. This superiority stems from the strategic allocation of workloads during the initialization phase, directly impacting execution runtime.

By preserving the original dataset structure, Index Bucketing eliminates the need for dataset reacquisition during subsequent executions. For instance, considering a scenario where the flattening process is repeated 100 times for each implementation, Index Bucketing showcases substantial performance superiority. Although multiple iterations of flattening might not align with typical real-world scenarios, this comparison demonstrates Index Bucketing's exceptional efficiency in executing additional feature implementations beyond flattening. Tasks like conditional filtering or attribute selection can be executed notably more efficiently with Index Bucketing compared to other implementations. This robust per-

formance exemplifies the enduring advantages of the Index Bucketing approach in handling repetitive operations and processing complex tasks swiftly.

Conclusion

Numerous strategies have been devised to tackle intricate challenges inherent in the manipulation of nested data structures. These complexities often stem from the presence of irregular schema, skewed distribution, information loss, and duplication explosion when performing queries on relational databases. Existing approaches addressing these issues necessitate computationally intensive mechanisms and the confines of traditional relational database environments inherently lack direct support for nested data structures. Index Bucketing introduces a novel framework to addressing aggregation and flattening stages in recent relational database methodologies and can be easily supported by document-oriented NoSQL database systems. The work explores an intuitive approach for mitigating irregular schema, skewed distribution information loss, and duplication explosion challenges. The efficacy of the proposed approach is assessed on prominent datasets such as SQuAD, QuAC, and NewsQA, comparing its performance against a competative Pandas implementation and a basic recursive, iterative implementation. Index Bucketing compares favorably in performance to these techniques, exemplifying the enduring advantages of Index Bucketing in handling repetitive operations and processing complex tasks. More insights can be gleaned from further evaluations expanding to other datasets and implementations. Future work will focus on extending the scope of the Index Bucketing algorithm and framework as well as evaluating parallel and distributed implementations.

REFERENCES

[1] A. B. Alexandrov et al. "Implicit Parallelism through Deep Language Embedding". In: *SIGMOD Rec.* 45 (2016), pp. 51–58.

[2] M. Armbrust et al. "Spark SQL: Relational Data Processing in Spark". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. Melbourne, Victoria, Australia, 2015.

[3] D. F. Bacon et al. "Spanner: Becoming a SQL System". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. Chicago, Illinois, USA, 2017.

[4] P. Carbone et al. "Apache flink: Stream and batch processing in a single engine". In: *The Bulletin of the Technical Committee on Data Engineering* 38.4 (2015).

[5] J. Cheney, S. Lindley, and P. Wadler. "Query Shredding: Efficient Relational Evaluation of Queries over Nested Multisets". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. 2014, pp. 1027–1038. DOI: 10.1145/2588555.2612186.

[6] S. Chlyah et al. "On the Optimization of Iterative Programming with Distributed Data Collections". In: (2022). Retrieved from https://inria.hal.science/hal-02066649.

[7] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (2008), pp. 107–113.

[8] R. Diestelkämper. "Explaining Existing and Missing Results over Nested Data in Big Data Analytics Systems". In: (2021).

[9] L. Fegaras. "An Algebra for Distributed Big Data Analytics". In: *Journal of Functional Programming* 27 (2017), e27.

[10] L. Fegaras. "Compile-time query optimization for Big Data analytics". In: *Open Journal of Big Data (OJBD)* 5.1 (2019), pp. 35–61.

[11] L. Fegaras and D. Maier. "Optimizing Object Queries Using an Effective Calculus". In: *ACM Trans. Database Syst.* 25.4 (2000), pp. 457–516.

[12] L. Fegaras and M. H. Noor. "Compile-time code generation for embedded data-intensive query languages". In: *2018 IEEE International Congress on Big Data*. July 2018.

[13] E. Meijer, B. Beckman, and G. Bierman. "Linq: reconciling object, relations and xml in the. net framework". In: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 2006.

[14] *MongoDB*. `https://www.mongodb.com/`. Accessed in November 2023.

[15] D. M. Quan Wang. *Algebraic Unnesting for Nested Queries: Oregon Graduate Institute School of Science & Engineering*. Tech. rep. 1999.

[16] W. Ricciotti and J. Cheney. "Mixing set and bag semantics". In: *Proceedings of the 17th ACM SIGPLAN International Symposium on Database Programming Languages*. Phoenix, AZ, USA, 2019. DOI: 10.1145/3315507.3330202.

[17] W. Ricciotti and J. Cheney. "Query Lifting: Language-integrated query for heterogeneous nested collections". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2021, pp. 579–606. ISBN: 9783030720193. DOI: 10.1007/978-3-030-72019-3_21. URL: http://dx.doi.org/10.1007/978-3-030-72019-3_21.

[18] B. Samwel et al. "F1 query: declarative querying at scale". In: *Proc. VLDB Endow.* 11.12 (2018), pp. 1835–1848.

[19] J. Smith. "Declarative nested data transformations at scale and biomedical applications". In: 2021. URL: https://api.semanticscholar.org/CorpusID:252163669.

[20] J. Smith et al. "Scalable querying of nested data". In: *arXiv preprint arXiv:2011.06381* (2020).

[21] J. Smith et al. "TraNCE: transforming nested collections efficiently". In: *Proc. VLDB Endow.* 14.12 (2021), pp. 2727–2730.

[22] D. Suciu. “Parallel programming languages for collections”. In: (1995).

[23] A. Ulrich. *Query Flattening and the Nested Data Parallelism Paradigm*. Tech. rep. Universität Tübingen, 2019.

[24] J. Van den Bussche. “Simulation of the nested relational algebra by the flat relational algebra, with an application to the complexity of evaluating powerset algebra expressions”. In: *Theoretical Computer Science* 254.1-2 (2001), pp. 363–377.

Copyright Permission

Name: Myers, Jeffrey

Email (to receive future readership statistics): jeffrey.myers648@topper.wku.edu

Type of document: ['Thesis']

Title: Index Bucketing: A Novel Approach to Manipulating Data Structures

Keywords (3-5 keywords not included in the title that uniquely describe content): Irregular Schema, Skewed Distribution, Information Loss, Duplication Explosion

Committee Chair: Yaser Mowafi

Additional Committee Members: Zhonghang Xia Huanjing Wang

Select 3-5 TopSCHOLAR® disciplines for indexing your research topic in TopSCHOLAR®: Theories and Algorithms Databases and Information Systems Computer Science

Copyright Permission for TopSCHOLAR® (digitalcommons.wku.edu) and ProQuest research repositories:

I hereby warrant that I am the sole copyright owner of the original work.
I also represent that I have obtained permission from third party copyright owners of any material incorporated in part or in whole in the above described material, and I have, as such identified and acknowledged such third-part owned materials clearly. I hereby grant Western Kentucky University the permission to copy, display, perform, distribute for preservation or archiving in any form necessary, this work in TopSCHOLAR® and ProQuest digital repository for worldwide unrestricted access in perpetuity. I hereby affirm that this submission is in compliance with Western Kentucky University policies and the U.S. copyright laws and that the material does not contain any libelous matter, nor does it violate third-party privacy. I also understand that the University retains the right to remove or deny the right to deposit materials in TopSCHOLAR® and/or ProQuest digital repository.

['I grant permission to post my document in TopSCHOLAR and ProQuest for unrestricted access.']

The person whose information is entered above grants their consent to the collection and use of their information consistent with the Privacy Policy. They acknowledge that the use of this service is subject to the Terms and Conditions.

['I consent to the above statement.']