LakeHarbor: Making Structures First-Class Citizens in Data Lakes

Hiroyuki Yamada
The University of Tokyo
Tokyo, Japan
hiroyuki@tkl.iis.u-tokyo.ac.jp

Masaru Kitsuregawa
The University of Tokyo
Tokyo, Japan
kitsure@tkl.iis.u-tokyo.ac.jp

Kazuo Goda
The University of Tokyo
Tokyo, Japan
kgoda@tkl.iis.u-tokyo.ac.jp

***Abstract*—This paper introduces *LakeHarbor*, a new data management paradigm that makes structures (e.g., indexes) first-class citizens in data lakes. The LakeHarbor paradigm enables a data lake system to flexibly construct structures based on registered access method functions and execute data processing jobs efficiently with the potential parallelism that the structures inherently hold by exploiting the functions while not sacrificing flexible data processing such as schema-on-read. This paper also presents *ReDe*, a prototype data processing engine that implements LakeHarbor, and a motivating evaluation and a case study of ReDe to explore the potential of LakeHarbor.**

I. Introduction

Data lakes have been a practical data management architecture in enterprises to deal with a vast amount of data with various data models and complex schemas. Data lake systems [11], [15] typically hold raw datasets and employ the schema-on-read processing model or hold datasets in open file formats such as Apache Parquet [14] to manage complex data schemas (e.g., nested columns).

Although data lakes achieve great flexibility in managing various data models and complex schemas, their data processing efficiency is not necessarily optimized, especially in selective data processing, due to the conservative exploitation of structures (e.g., indexes). Enterprises move a curated subset of their data into data warehouses to achieve high performance for demanding structured data access workloads. However, the two-tier architecture is complex and causes data freshness problems; data in data warehouses is likely to be out of sync with the raw data in data lakes due to the time-consuming extract, transform, and load (ETL) process.

Several approaches [10], [34] exist for creating structures directly on top of data lakes to mitigate such efficiency issues while avoiding data freshness problems. These approaches are designed to process data based on the processing model applied in the underlying data lake systems; hence, they necessarily provide limited capabilities and expressibility for structured data accesses. For example, they execute selective data processing with dozens of statically defined parallelism (usually matching the number of CPU cores) in each computing node, and they also cannot execute parallel nested loop joins with global indexes [38] to improve selective data processing performance.

This paper introduces *LakeHarbor*, a new data management paradigm that makes structures (e.g., indexes) first-class citizens in data lakes. The LakeHarbor paradigm enables a data lake system to flexibly construct structures based on registered access method functions and execute data processing jobs efficiently with the potential parallelism that the structures inherently hold by exploiting the functions while not sacrificing the great flexibility of data lakes, e.g., flexible data processing with schema-on-read.

There are several other efforts to increase data processing efficiency while retaining the flexibility of data lakes. Lakehouse [45] is a data management architecture designed to take the benefits of both a data warehouse and a data lake. Lakehouse aims to improve data processing efficiency by exploiting auxiliary data and optimizing data layout while keeping open file formats such as Apache Parquet. Self-organizing data container (SDC) [30] provides a storage format specialized for cloud-based data lakes. SDC organizes a container composed of multiple data files with rich metadata and adaptively optimizes the data layout of the container to improve data processing efficiency. LakeHarbor is complementary to these data management architectures but takes a more drastic approach in terms of exploiting structures and their potential parallelism.

This paper also presents *ReDe*, a prototype data processing engine that implements the LakeHarbor paradigm. ReDe provides an abstraction called *Reference-Dereference* for defining a data processing job with access methods. Specifically, Reference-Dereference composes a data processing job with a list of sequential *reference* and *dereference* functions. The abstraction is based on the fact that a wide variety of data processing jobs can be expressed with a list of obtaining a pointer from a record (referencing) and obtaining a record from a pointer (dereferencing). This abstraction derives the structural information of data and the data dependencies about data accesses, which enable ReDe to construct structures flexibly and execute data processing jobs efficiently with the potential parallelism derived from the structures. Consequently, ReDe executes data processing jobs with massive parallelism, which could bring significant performance improvement, especially for selective data processing.

This paper makes the following contributions:

- We introduce LakeHarbor, a new data management paradigm that enables a data lake system to flexibly con-

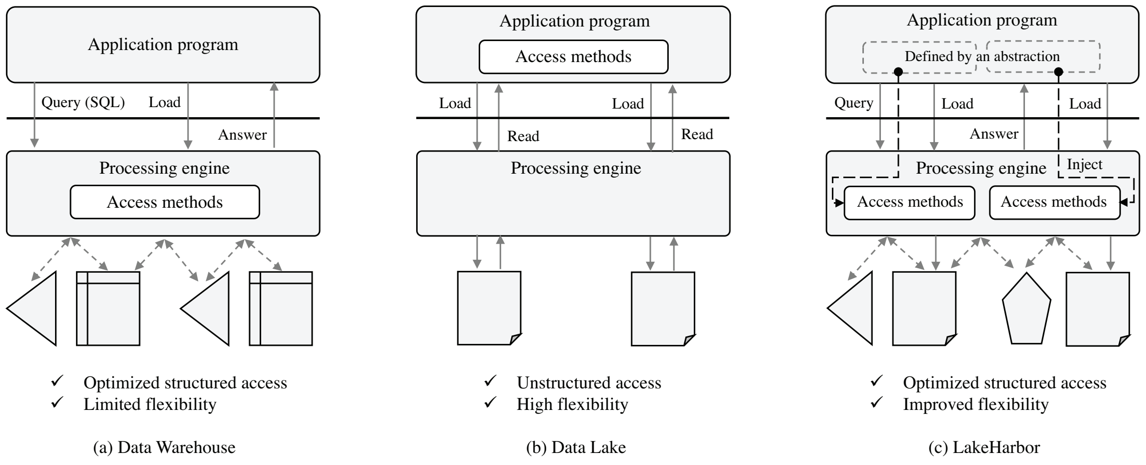


Fig. 1: Comparison between data management paradigms: data warehouses, data lakes, and LakeHarbor.

struct structures and execute data processing jobs efficiently with the potential parallelism derived from the structures without sacrificing the flexible data processing of data lakes.

- We also present ReDe, a prototype data processing engine that implements LakeHarbor, and a motivating evaluation and a case study of ReDe to explore the potential of LakeHarbor.
- We describe a number of research directions for LakeHarbor.

The remainder of the paper is organized as follows. Section II discusses our motivation and vision for LakeHarbor. Section III introduces ReDe, a prototype data processing engine that implements LakeHarbor, and a motivating evaluation of ReDe. Section IV describes a case study of ReDe. Section V discusses some of the new problems and opportunities that LakeHarbor present. Section VI describes related work. Finally, Section VII concludes the paper.

II. LakeHarbor: Motivation and Vision

Data warehouses (Figure 1(a)) have been a mainstream solution for enterprises to store and manage large-scale data. Data warehouses organize given data in a form optimized for expected query workloads. This approach helps to improve query performance. Instead, it incurs significant performance and capacity overheads for loading new data and sometimes yields undesirable performance for unexpected workloads. Data warehouses typically use relational database systems to store and query data; thus, record schemas (e.g., record format) and structures (e.g., indexes, logical relationships between tables) are pre-defined. In short, the data processing engine of data warehouses defines access methods based on the relational model. This leads to inflexible data processing; i.e., data warehouses cannot efficiently store and query data with complex schemas or unstructured data that do not fit well with the relational model.

Data lakes (Figure 1(b)) are an emerging approach to reduce the organizing efforts indispensable for data warehouses. Data lakes opt to store given data in a raw form; thus, they remove the organizing overheads for loading new data in terms of performance and capacity, and interpret the schemas and structures of data on the fly on the applications flexibly when reading the data. In other words, applications define access methods. Enterprises can invest the reduced cost to extend the system scalability so that data lakes can offer reasonable performance for expected and unexpected query workloads. These technical properties are widely accepted by many, often burgeoning, enterprises that must be agile and flexible in responding to changes in their business environment.

Obviously, the data lake solution misses the technological opportunity that organizing data improves the performance for expected query workloads. Balanced solutions between the two extremes, data warehouses and data lakes, are actively studied. Lakehouse [45] aims to improve data processing efficiency by exploiting auxiliary data and optimizing data layout while keeping open file formats such as Apache Parquet. A self-organizing data container (SDC) [30] organizes a container composed of multiple data files with rich metadata and adaptively optimizes the data layout of such container to improve data processing efficiency. Several approaches [10], [34] also exist for organizing structures directly on top of data lakes to improve query performance for expected query workloads. Although the existing balanced solutions offer good performance improvement, they still employ conservative approaches for dealing with structures in data lakes. For example, they execute selective data processing with dozens of statically defined parallelism (usually matching the number of CPU cores) in each computing node, and they also cannot execute parallel nested loop joins with global indexes to improve selective data processing performance.

LakeHarbor (Figure 1(c)) is our balanced solution with a more drastic approach in terms of exploiting structures in data lakes. LakeHarbor enables the post hoc definition of access methods for data stored in data lakes; the user or the third-party software is allowed to inject access method definitions that describe how one can interpret and access target data. LakeHarbor then creates auxiliary data structures (e.g., indexes) for the target data, if necessary, by using the definitions and uses the structures to access the data efficiently. Since the access method definitions could contain arbitrary logic, users can create structures flexibly even on complex schemas, such as nested columns, and could also exploit the structures when accessing the data by, for example, using nested loop joins with global indexes while not spoiling the flexibility properties (e.g., schema-on-read) that data lakes hold by nature. Moreover, LakeHarbor obtains detailed structural information of the stored data through the access method definitions; thus, it could execute data processing jobs efficiently with the potential parallelism that the structures inherently hold. Consequently, LakeHarbor could execute data processing jobs with fine-grained massive parallelism, which could bring significant performance improvement, especially for selective data processing. Fine-grained massively parallel execution [17], [28], [43], which fully exploits the parallelism of modern hardware that could handle more than thousands of concurrent IOs, has been recently explored and validated in relational query processing. LakeHarbor is a solution that achieves fine-grained massive parallelism in data lakes without compromising the benefits of data lakes.

We have experienced several real-world workloads that LakeHarbor systems would significantly benefit, as described in Section IV.

III. ReDe

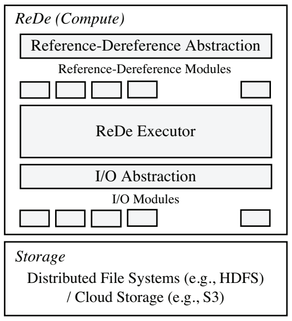
This section introduces ReDe, a prototype data processing engine (query engine) that implements the LakeHarbor paradigm. There could be many design choices for implementing LakeHarbor, and ReDe is one implementation to clarify the potential of LakeHarbor.

A. System Overview

ReDe is a distributed data processing engine designed based on the *separation of compute and storage* architecture, which has been employed in many recent query engines [3], [40], [41]. ReDe accepts a job (or a query) written with its abstraction called *Reference-Dereference* and executes the job with its executor called ReDe Executor. ReDe Executor uses I/O abstraction, which abstracts underlying storage implementations, to separate compute and storage. Figure 2(a) shows the architecture of ReDe.

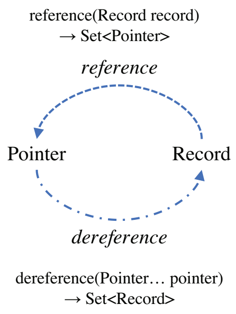
B. Reference-Dereference

Reference-Dereference (Figure 2(b)) is an abstraction for defining a data processing job with access methods. Specifically, Reference-Dereference composes a data processing job with a list of sequential *reference* and *dereference* functions. A *reference* function takes a record and produces a set of pointers to other records that the record is associated with. A *dereference* function takes a pointer or two pointers and produces a set of records that the pointer points to or a set of



ReDe (Compute)
Reference-Dereference Abstraction
Reference-Dereference Modules
ReDe Executor
I/O Abstraction
I/O Modules
Storage
Distributed File Systems (e.g., HDFS) / Cloud Storage (e.g., S3)

(a) Architecture.



reference(Record record) → Set<Pointer>
reference
Pointer
Record
dereference
dereference(Pointer… pointer) → Set<Record>

(b) Reference-Dereference.

Fig. 2: ReDe.

records between the ranges that the two pointers point to. The abstraction is based on the fact that a data processing job can be expressed with a list of obtaining a pointer from a record (*referencing*) of a file and obtaining a record of a file from a pointer (*dereferencing*) in many cases.

Reference-Dereference uses I/O abstraction and concrete I/O module implementations for the abstraction to access the underlying data of each storage. I/O abstraction defines three basic interfaces: *Record*, *Pointer*, and *File*. A *Record* is a unit of data that ReDe reads and writes. A *Pointer* is a logical (e.g., record's primary key) or physical (e.g., file offset) pointer used to locate a *Record*. A set of *Record*s composes a *File*. *File* is assumed to be distributed into partitions and can locate a *Record* with the corresponding *Pointer*. Therefore, a *Pointer* also contains partition information to properly locate a *Record*. Specifically, a *File* takes a partition key from a given *Pointer*, applies it to a pre-configured Partitioner (e.g., HashPartitioner or RangePartitioner) to locates a partition, and locate a *Record* with an in-partition key that can also be taken from the *Pointer*. There is also a special *File* called *BtreeFile*. A *BtreeFile* can also locate a set of *Record*s with a range of given *Pointer*s.

To clarify how Reference-Dereference works, consider a job of join processing for the Part and Lineitem files of the TPC-H dataset. We assume the Part file is hash-partitioned by p_partkey and the Lineitem file is hash-partitioned by l_orderkey. There are also B-tree indexes on p_retailprice and l_partkey that are hash-partitioned by p_partkey and l_partkey, respectively. Note that the files are managed with *File*, and the B-tree indexes are managed with *BtreeFile*. The job corresponds to the following SQL:

```
SELECT * FROM Part p JOIN Lineitem l
ON p.p_partkey = l.l_partkey
WHERE p.p_retailprice BETWEEN X AND Y
```

Figure 3 shows how the join processing can be expressed as a list of reference and dereference functions, called *Referencers* and *Dereferencers*. Figure 4 shows the (simplified) Java code of the *Referencers* and *Dereferencers* for the join processing.

We first look at how the functions obtain a Part record from its index on p_retailprice values. The first function

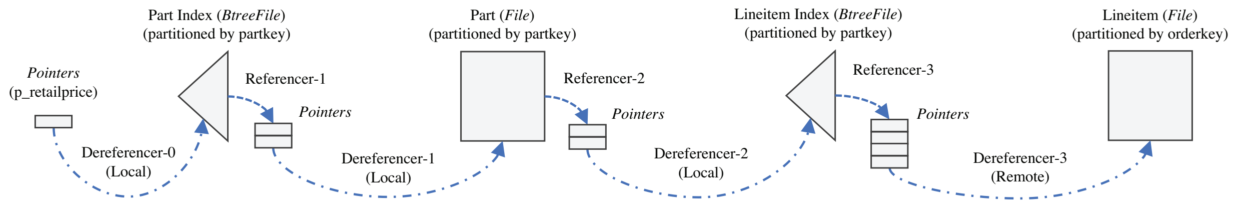
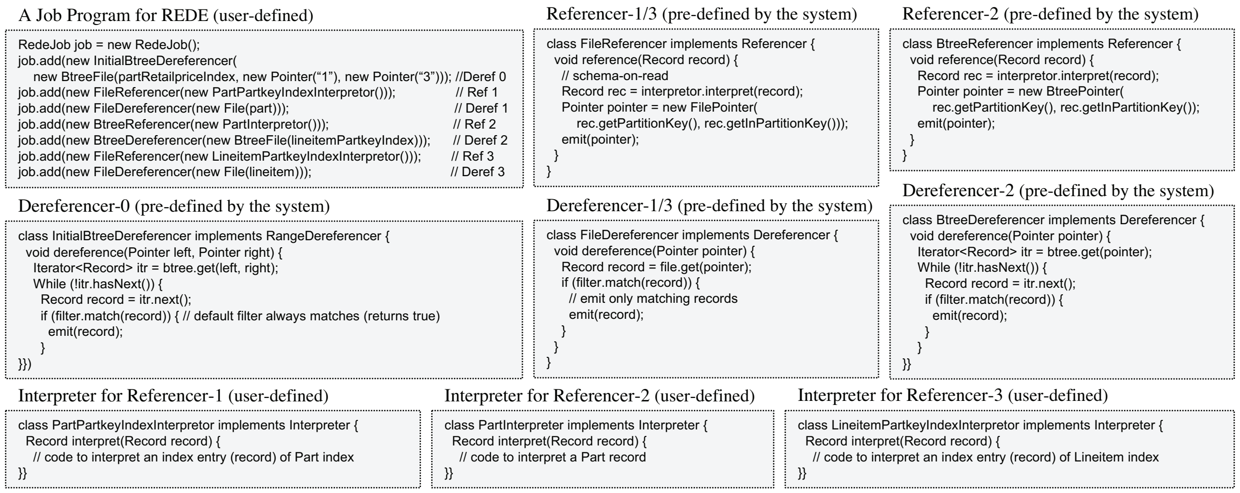


Fig. 3: Example *Referencers* and *Dereferencers* for Part-Lineitem join.


A Job Program for REDE (user-defined)

```
RedeJob job = new RedeJob();
job.add(new InitialBtreeDereferencer(
   new BtreeFile(partRetailpriceIndex, new Pointer(“1”), new Pointer(“3”))); //Deref 0
job.add(new FileReferencer(new PartPartkeyIndexInterpretor()));             // Ref 1
job.add(new FileDereferencer(new File(part)));                              // Deref 1
job.add(new BtreeReferencer(new PartInterpretor()));                        // Ref 2
job.add(new BtreeDereferencer(new BtreeFile(lineitemPartkeyIndex)));        // Deref 2
job.add(new FileReferencer(new LineitemPartkeyIndexInterpretor()));         // Ref 3
job.add(new FileDereferencer(new File(lineitem)));                          // Deref 3
```

Referencer-1/3 (pre-defined by the system)

```
class FileReferencer implements Referencer {
  void reference(Record record) {
    // schema-on-read
    Record rec = interpretor.interpret(record);
    Pointer pointer = new FilePointer(
       rec.getPartitionKey(), rec.getInPartitionKey()));
    emit(pointer);
  }
}
```

Referencer-2 (pre-defined by the system)

```
class BtreeReferencer implements Referencer {
  void reference(Record record) {
    Record rec = interpretor.interpret(record);
    Pointer pointer = new BtreePointer(
       rec.getPartitionKey(), rec.getInPartitionKey());
    emit(pointer);
  }
}
```

Dereferencer-0 (pre-defined by the system)

```
class InitialBtreeDereferencer implements RangeDereferencer {
  void dereference(Pointer left, Pointer right) {
    Iterator<Record> itr = btree.get(left, right);
    While (!itr.hasNext()) {
      Record record = itr.next();
      if (filter.match(record)) { // default filter always matches (returns true)
        emit(record);
      }
}})
```

Dereferencer-1/3 (pre-defined by the system)

```
class FileDereferencer implements Dereferencer {
  void dereference(Pointer pointer) {
    Record record = file.get(pointer);
    if (filter.match(record)) {
      // emit only matching records
      emit(record);
    }
  }
}
```

Dereferencer-2 (pre-defined by the system)

```
class BtreeDereferencer implements Dereferencer {
  void dereference(Pointer pointer) {
    Iterator<Record> itr = btree.get(pointer);
    While (!itr.hasNext()) {
      Record record = itr.next();
      if (filter.match(record)) {
        emit(record);
      }
    }
}}
```

Interpreter for Referencer-1 (user-defined)

```
class PartPartkeyIndexInterpretor implements Interpreter {
  Record interpret(Record record) {
    // code to interpret an index entry (record) of Part index
}}
```

Interpreter for Referencer-2 (user-defined)

```
class PartInterpreter implements Interpreter {
  Record interpret(Record record) {
    // code to interpret a Part record
}}
```

Interpreter for Referencer-3 (user-defined)

```
class LineitemPartkeyIndexInterpretor implements Interpreter {
  Record interpret(Record record) {
    // code to interpret an index entry (record) of Lineitem index
}}
```

Fig. 4: Code of *Referencers* and *Dereferencers* for Part-Lineitem join.

Dereferencer-0 takes a range of Part.p_retailprice values as arguments and uses the B-tree index to get a set of matching records. Note that every *Dereferencer* manages either a *File* or a *BtreeFile* to access. In this example, *Dereferencer-0* manages a local secondary B-tree index of the Part file, and the obtained records consist of logical pointers of the Part file. It then emits each record if the record matches a filtering condition. The filtering condition is optionally provided at the time of job definition with a function called *Filter*, which interprets a given record with schema-on-read and filters out the record if the given condition does not match the record.[1] Following that, the second function *Referencer-1* takes the record that was emitted in the previous dereference function. It also interprets the schema of the record with a function called *Interpreter* , which interprets a given record with schema-on-read. It then creates a pointer to a Part record from the interpreted record and emits the pointer. As we explained, the pointer is not necessarily pointing to a local file; thus, it has a partition key to locate a node. The third function *Dereferencer-1* takes the pointer and accesses the Part file using the pointer to get the corresponding record.

Next, we look at how the functions obtain a Lineitem record from the obtained Part record. *Referencer-2* takes the Part record and extracts a pointer to the B-tree index of Lineitem.l_partkey. *Dereferencer-2* takes the pointer and uses the B-tree index to get a set of matching records. *Referencer-3*, the same code as *Referencer-1*, creates a pointer to the Lineitem file in the same way. *Dereferencer-3*, which is the same code as *Dereferencer-1*, accesses the Lineitem file using the pointer. Note that the last dereferencing fetches the Lineitem records through cross-partition accesses since the index for l_partkey and Lineitem are partitioned by different keys.

A ReDe job defines a list of the reference and dereference functions, as described in Figure 4. Composing such a list is similar to creating a MapReduce [6] job caring for how data is partitioned.

The Reference-Dereference abstraction is conceptually similar to Skywriting in CIEL [31], [32], which manages data dependencies of data by applying a dereference operator to a data reference in a program. However, Skywriting is designed for managing coarse-grained data such as files; thus, it cannot extract the structures of data, i.e., it cannot exploit the fine-grained parallelism of the data. On the other hand, Reference-Dereference is designed to extract fine-grained data parallelism, which could bring significant performance benefits, as discussed in Section III-C.

Although the abstraction seems to offer a limited or low-level programming interface, we believe it is still suitable for

[1]No filter is given in the example for ease of explanation.

a broad class of applications and not very difficult to manage for programmers.

Expressibility. Reference-Dereference can express a wide range of index-based structured data processing such as selection and join. Specifically, the current design supports data processing using the indexing schemes defined in a taxonomy paper [38]. For example, as described in Figure 4, it can express parallel index nested loop joins whether or not the used indexes are local or global. Moreover, it can express broadcast joins, where index pointers are broadcasted to all the partitions. Specifically, the broadcast joins can be expressed by passing a null value to the partition information of the pointer emitted by a *Referencer*, which makes the system replicate the given pointer to all the partitions. Multi-way joins can also be naturally expressed by appending *Referencers* and *Dereferencers*.

Usability. *Referencers* and *Dereferencers* to support the indexing schemes [38] are pre-defined by the system and reusable. For example, all the *Referencers* and *Dereferencers* in Figure 4 are pre-defined except for *Interpreters*. Therefore, programmers' task to define a job in most cases is choosing *Referencers* and *Dereferencers* to use, creating an *Interpreter* for each *Referencer* for schema-on-read, optionally creating a *Filter* for each *Dereferencer*, and composing a list of *Referencers* and *Dereferencers* that makes sense as a data processing, which is similar to creating a MapReduce job caring for how data is partitioned. Even if programmers want to create complex *Referencers* and *Dereferencers* that are not pre-defined, they need to create them only for each file, not for each job, since the functions are not specific to jobs, as seen in Figure 4. Thus, we believe that Reference-Dereference does not give too much burden to programmers, and programmers who have experience in MapReduce or Spark can create a ReDe job without much difficulty.

C. *Optimizing Execution Efficiency*

The Reference-Dereference abstraction helps to derive the structural information of the data accessed by a job and the data dependencies about the data accesses. ReDe leverages the information and data dependencies to dynamically decompose a job into fine-grained tasks during job execution and achieves scalable massively parallel execution (SMPE) [43] in data lakes.

To clarify how ReDe's SMPE works, consider the same job (query) described in the previous section in a three-node cluster.[2] Figure 5 illustrates the simplified form of ReDe execution for nested loop joins between Part and Lineitem.

ReDe is initiated by distributing the data processing job to all the computing nodes. For each node, it first retrieves a pointer given from the job and passes it to *Dereferencer-0*. Assuming that there is a certain degree of cardinality in the index and the condition predicate matches with several distinct values; thus, it emits multiple pointers of Part (three in the figure). When fetching the records of Part with the pointers with *Dereferencer-1*, ReDe creates a thread for each dereference function invocation and uses the thread to fetch the corresponding record of Part by exploiting the independence of record accesses. For each fetched record of Part, it can extract the foreign key attribute with *Referencer-2* to access Lineitem. The extraction for multiple records could also be done in parallel by creating a thread for each referencer function invocation.

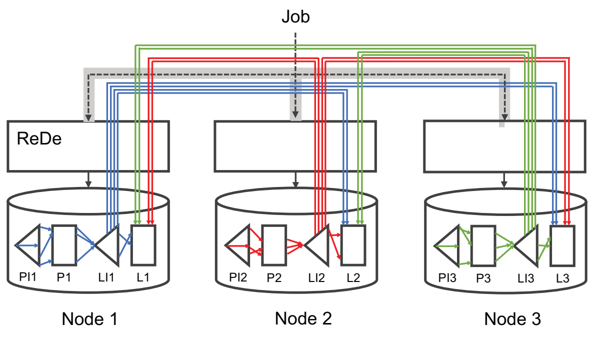


Fig. 5: Example of ReDe execution for nested loop joins.

Once foreign keys are extracted, it accesses the index of Lineitem with *Dereferencer-2*. Since the accesses are independent as well, multiple dereference function invocations can be executed in parallel in the same way. Here we assume that it produces more pointers than the number of foreign keys due to a fanout (each key producing two pointers in the figure; thus, six pointers). Then, for each obtained pointer, each node accesses Lineitem with *Dereferencer-3*. Similarly, these accesses can be done in parallel whether or not they are local accesses or remote accesses. As illustrated in Figure 5, all the nodes execute the data processing job in the same way.

As can be seen, the job has a lot more parallelism than the partitioned parallelism given by the three nodes. ReDe potentially exploits the parallelism derived from a given job and the underlying nodes' hardware capacity, such as IOPS, at full. Note that we explain the approach in a simplified form; however, there will be more opportunities for parallel processing. For example, when a data processing job is N-way join where N is bigger than two, it could execute with more parallelism because it accesses more records.

Figure 6 shows the execution model of ReDe for achieving SMPE and Algorithm 1 shows the SMPE algorithm based on the model. ReDe divides a data processing job into multiple stages and executes one of the given functions (i.e., *Referencer* and *Dereferencer*) in each stage. Each stage has an input queue and an output queue, and the output queue of one stage is the input queue of the next stage.

As discussed, ReDe is initiated by distributing a data processing job to all the computing nodes (lines 2-5), and each node executes the stages of the job (lines 8-18). In each node, SMPE takes the *Dereferencer* function of the

[2]We explain the example with a shared-nothing cluster for ease of explanation, but ReDe works the same even on a shared disk such as S3.

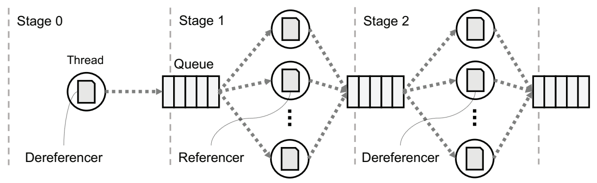


Fig. 6: Execution model of ReDe for SMPE.

initial stage, executes the function (lines 14, 19-24, and 45), and puts the emitted outputs of the function into the queue (lines 47-51). Note that ReDe executes the initial stage and the *Dereferencer* function on different threads from the main thread not to block the execution of other stages and functions. Then, ReDe creates threads and executes the other *Referencer* and *Dereferencer* functions of the subsequent stages on the threads whenever it detects data (i.e., a pointer or a record) in the queue (lines 16, 25-42). Specifically, it creates two threads for each data from the queue, and dispatches one to execute a function (lines 44-45) and dispatches the other to handle the emitted outputs of the function (lines 47-51) so that executing the function does not block the execution of other stages and functions. Consequently, ReDe executes the functions massively in parallel.

In the current implementation, ReDe manages threads in a thread pool and reuses them instead of creating them every time. It manages 1000 threads in the default setting, but the number can be adjusted based on underlying hardware capabilities such as the number of CPU cores and the IOPS of IO path. Moreover, as an optimization, ReDe does not switch threads for *Referencers* by default to avoid excessive context switching because *Referencers* do not usually incur IO and are lightweight.

D. Structure Maintenance

ReDe builds indexes flexibly in the background by using registered *Interpreter*s and *Referencer*s. An *Interpreter* for a *File* extracts a partition key and an index key in the partition from each record, and a *Referencer* emits a pair of the partition key and the index key for the record. Then, ReDe lazily creates indexes by using the emitted pair. With the mechanism, ReDe provides a flexible indexing scheme even on complex schemas, as described in Section IV.

E. Preliminary Evaluation

This section evaluates one of the benefits of ReDe, the efficiency of structured data processing, to clarify the potential of LakeHarbor.

Environment. We ran our experiments on a 128-node cluster. Each node was a 2U server that was equipped with two Intel Xeon E5-2680 2.70 GHz processors (16 cores in total), 64 GB memory, two 15K RPM 300 GB SAS HDDs for OS (OS drives), and twenty-four 10K RPM 900 GB SAS HDDs for data (data drives). We created a RAID-6 array across the twenty-four HDDs using the attached RAID controller. Each node ran CentOS Linux and used ext4 for both drives.

Algorithm 1 Scalable Massively Parallel Execution in ReDe.

```
 1: function EXECUTESMPE($job$, $nodes$)
 2:     for $i \leftarrow 1, length(nodes)$ do
 3:         // invocation returns immediately before completion
 4:         INVOKE($nodes[i]$, EXECUTESMPEEACH)
 5:     end for
 6:     Wait until all the execution is completed
 7: end function
 8: function EXECUTESMPEEACH($job$)
 9:     $queue \leftarrow$ CREATEQUEUE
10:     // the order of $funcs$ specifies data dependencies,
11:     // and $funcs$ define structural information
12:     $funcs \leftarrow$ GETFUNCTIONS($job$)
13:     $t1 \leftarrow$ CREATETHREAD
14:     EXECUTEINITIALSTAGE($funcs$, $queue$) on $t1$
15:     $t2 \leftarrow$ CREATETHREAD
16:     EXECUTESTAGES($funcs$, $queue$) on $t2$
17:     Wait until the execution on the threads are completed
18: end function
19: function EXECUTEINITIALSTAGE($funcs$, $queue$)
20:     $stage \leftarrow 0$
21:     $func \leftarrow funcs[stage]$    ▷ $func$ is the initial Dereferencer
22:     $input \leftarrow$ GETINPUT($func$)
23:     EXECUTEFUNC($func$, $input$, $queue$)
24: end function
25: function EXECUTESTAGES($funcs$, $queue$)
26:     while until all tasks are finished do
27:         $input \leftarrow$ DEQUE($queue$)
28:         if $input$ does not have partition information then
29:             SETPARTITION($input$, LOCAL)
30:             BROADCAST($input$)
31:                             ▷ enque $input$ to all the nodes' queues
32:             continue
33:         end if
34:         $func \leftarrow funcs[input.stage]$
35:                             ▷ $func$ is Referencer or Dereferencer
36:         if $func$ is null then
37:             continue
38:         end if
39:         $t \leftarrow$ CREATETHREAD  ▷ create if $func$ is Dereferencer
40:         EXECUTEFUNC($func$, $input$, $queue$) on $t$
41:     end while
42: end function
43: function EXECUTEFUNC($func$, $input$, $queue$)
44:     $t \leftarrow$ CREATETHREAD       ▷ create if $func$ is Dereferencer
45:     $func(input)$ on $t$            ▷ outputs are pushed to $emitted$
46:                         ▷ $func$ might fetch data from remote nodes
47:     while until all emitted results are processed do
48:         $output \leftarrow$ DEQUE($emitted$)
49:         $new\_input \leftarrow$ CREATEINPUT($output$, $input.stage+1$)
50:         ENQUE($queue$, $new\_input$)
51:     end while
52: end function
```

For the data drive, we set noop for IO scheduler and 1008 for nr_request and queue_depth parameters.The nodes were connected with Dell Force10 Z9000 10 Gbps switch.

As a preliminary evaluation, we compared ReDe with a fast data lake system, Apache Impala [13]. Impala is a query engine focusing on analytical workloads and not supporting indexes. We used Impala version 3.0. In future work, we will conduct more experiments with other query engines, such as Spark [15], Photon [3], and Snowflake [41].

Dataset. We used TPC-H dataset [5] for the experiments. We generated files with SF=128K. The total size of all the files was about 128TB. We created a HDFS [11] cluster using the data drives of the nodes and loaded the dataset into the HDFS

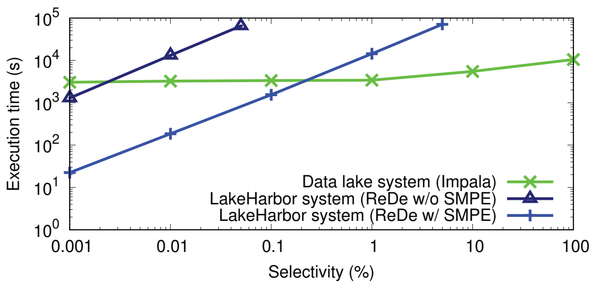


Fig. 7: Performance comparison between a data lake system and a LakeHarbor system (ReDe).

cluster where the dataset was distributed into the nodes by round-robin.

For ReDe, we created a simple distributed file system for the experiments and used it instead of HDFS since HDFS is not well-optimized for non-scan accesses such as lookups. We loaded the files into the distributed file system, which distributed the files into 128 partitions evenly spread into the nodes by hashing with their primary keys. We also created local secondary indexes on the date columns (e.g., o_orderdate in Order) of each file and global indexes for each foreign key of each file. Each global index is also distributed into partitions by the corresponding foreign key.

Workload. We used a simplified TPC-H query (TPC-H Q5'), which is a variant of the TPC-H Q5 query, where the sorting and aggregation are removed to focus on clarifying the performance differences for a SPJ (select-project-join) workload. We also varied the selectivities of the query using the predicates to cover a wide selectivity range to see how the systems behave for each selectivity.

Evaluation Results. Figure 7 shows the evaluation results. Impala executed the query using (grace) hash joins; the execution time of Impala gradually increased as the selectivity increased. On the other hand, ReDe executed the query using *Referencers* and *Dereferencers* (i.e., parallel nested loop joins); the execution time of ReDe increased more steeply as the selectivity increased. ReDe (w/o SMPE) simply used the created structures and the partitioned parallelism given from data partitions; thus, it showed a slight performance benefit over Impala in the very low selectivity range. By contrast, ReDe (w/ SMPE) outperformed Impala by more than an order of magnitude in a wide range of selectivities because of scalable massively parallel execution, which effectively exploited the derived structural information of data accesses and the data dependencies about the data accesses, as explained in Section III-C. Note that ReDe became slower than Impala in the high selectivity range because the current prototype does not implement efficient data processing on unstructured data or a query optimizer. If ReDe implements them, ReDe could choose data processing plans appropriately based on query selectivities; i.e., ReDe would perform comparably with Impala in the high selectivity range.

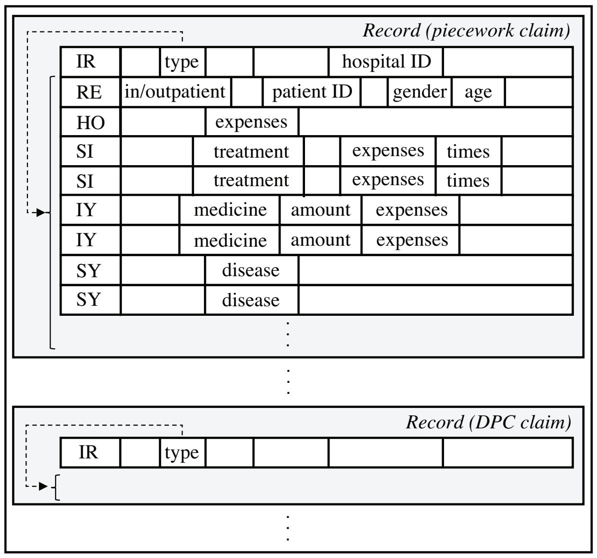


Fig. 8: Example of Japanese insurance claims.

IV. Case Study

In this section, we introduce a use case of ReDe. As discussed in Section II, the primary benefit of data lake is its flexibility in data formats. In contrast to data warehouses, which store data according to pre-defined schemas based on the relational model, data lakes do not pre-define the schemas of data and let the applications interpret them on the fly when reading the data. ReDe inherits the benefit of data lakes while making it possible to specify access method definitions in a post hoc manner. The access method definitions enable ReDe to obtain detailed structural information of stored data; thus, ReDe could execute data processing jobs efficiently with the potential parallelism that the data inherently holds.

Let us dive into a case of the analytics of public healthcare insurance claims in Japan. Similar to the UK and Germany, Japan has employed the universal service policy; publicly operated healthcare insurance programs cover all necessary medical care for all citizens and compensate for most medical expenses. Insurance claims are digitally managed and exchanged between medical institutions and public insurers, each describing medical expenses charged to a patient and its evidential information, such as diagnosed diseases and medical services (e.g., prescriptions and treatments) provided to the patient. The nationwide collective database of these insurance claims has a strong potential to gain a broad spectrum of new findings on medical policies and medical technologies.

The data format of the insurance claims, standardized by the government, contains a high structural complexity to describe various clinical situations, as shown in Figure 8. The insurance claims file is a text comprised of multiple records, each comprising multiple sub-records of different kinds. The format of each sub-record is determined by the two leading characters. "IR" indicates a record describing a hospital claiming the medical expenses. The type attribute of an IR sub-record specifies if the record is a piecework or a DPC claim; hence, the records are dynamically defined. A ”RE” sub-record describes a service category (e.g., in-patient or out-patient) and patient information. A ”HO” sub-record describes total medical expenses. ”SI”, ”IY”, and ”SY” sub-records describe medical treatments provided to the patient, medicines prescribed to the patient, and diseases diagnosed to the patient, respectively.

We tried two major approaches to analyzing the insurance claims: (1) normalizing the data based on the relational model and storing it in a data warehouse system that employs fine-grained massively parallel execution [17], and (2) storing it in a raw form in a data lake system. The first approach yielded performance penalties due to intensive joins of normalized data even though fine-grained massively parallel execution helped to improve overall performance. The second approach provided slow performance due to a full data scan with the statically defined parallelism based on the data lake system. Moreover, the second approach could not utilize nested-column file formats such as Apache Parquet [14] because such file formats cannot properly express the dynamically defined records of the insurance claims.

We then came up with ReDe as our solution, which stores insurance claims in raw form in storage and defines how the data is accessed. ReDe performed significantly better than the other systems by eliminating the performance overhead of joins while executing queries with massive parallelism.

The performance differences between ReDe and the data warehouse system mainly came from the differences in the number of record accesses. That is because both systems accessed their stored data with fine-grained massively parallel execution, and the number of record accesses determines the theoretical limitation of query performance in those systems. Figure 9 shows the normalized numbers of record accesses of these systems for the following three queries:[3]

Q1 Calculate medical expenses charged to medical care prescribing antihypertensive medicines for hypertension.

Q2 Calculate medical expenses charged to medical care prescribing antimicrobial medicines to acne patients.

Q3 Calculate medical expenses charged to medical care prescribing GLP-1 receptor medicines to diabetes patients.

The results show that while ReDe executed queries with fine-grained massively parallel execution, it accessed significantly fewer records because its flexible data processing with schema-on-read avoided the intensive joins caused by data normalization.

ReDe has been running in a real research platform for healthcare analyses by an interdisciplinary team between computer science and healthcare in Japan [18]. Specifically, it has been employed as a data analytics infrastructure of the research platform to analyze the nationwide insurance claims database and has provided an efficient data processing service to healthcare researchers [20], [21], [23]–[25], [35], [36], [39], [42].

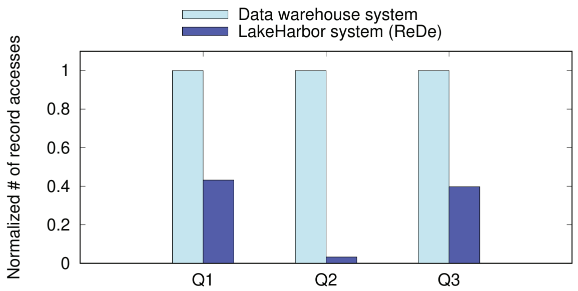


Fig. 9: Differences in the number of record accesses between a data warehouse system that employs fine-grained massively parallel execution and a LakeHarbor system (ReDe). The numbers are normalized based on the number of the data warehouse system.

This section has focused on a case of Japanese healthcare insurance analytics, but ReDe could be applied to other cases. The international medical community has recently promoted FHIR [16], the format standard of electronic medical records. FHIR has a similar design to the Japanese insurance claims format, employing the nested record organization. We expect ReDe would also manage and process the FHIR data flexibly and efficiently.

V. Research Directions

So far, we described the high level idea of LakeHarbor and its example implementation (ReDe) to verify the potential of LakeHarbor. In this section, we focus on some of the new problems and opportunities that LakeHarbor present.

A. Abstraction

We introduced one example of systems that support LakeHarbor, but there could be a system that better supports the paradigm. For example, the Reference-Dereference abstraction of ReDe can fully exploit structures and provide flexible and efficient data processing; however, as a trade-off, it might not be high-level enough. A higher-level abstraction brings not only better usability but also an opportunity for query optimizations, which could help to select appropriate structures for efficient data processing. Exploring higher-level abstractions without compromising flexibility and efficiency is an important research challenge.

B. Structure Maintenance

Although we have discussed the concept of registering access methods and lazily building structures using the access methods, we have not discussed what structures to build and at what times. It is essential to clarify those to build a practical LakeHarbor system, and we believe the following should be considered. First, having many structures could provide more opportunities to derive more efficient structured data processing; however, more structures could cause more

[3]We omitted the result of the data lake system because it was a lot slower than the others.

performance and capacity overheads for loading new data. Therefore, we should care about data processing performance and loading performance to decide what structures to build. Second, workloads are not static in recent analytics, so structure maintenance should be adaptive to workload changes and future workloads.

C. Storage Engine

Since systems for LakeHarbor fully exploit the parallelism of structures, their data access workloads could be more fine-grained than the ones of existing systems for data lakes and Lakehouses. Emerging storage engines such as Delta Lake [2] and Apache Iceberg [12] are promising approaches but not likely to be optimized enough for such workload. It is worth exploring a new storage layer for better efficiency in the LakeHarbor workload.

D. Integration with Existing Systems

LakeHarbor is not an exclusive paradigm and could coexist with other data management paradigms, such as Lakehouse. Therefore, integrating a LakeHarbor system and a Lakehouse system might be a practical and promising approach to achieve a system that gets the best of both worlds. This architecture opens up a new design space. First, how to integrate multiple systems based on different paradigms is challenging. Second, even if multiple systems are integrated seamlessly, it is challenging to optimize a query to derive an execution plan that best utilizes the benefits of all the systems.

VI. Related Work

A. Abstractions for Processing Large-scale Data

MapReduce [6], Dryad [22], Spark [44], and CIEL [32] provide general-purpose abstractions (or programming models) for processing large-scale data. Although these provide different abstractions, they are designed for achieving coarse-grained parallelism. ReDe is one of the abstractions for LakeHarbor and is specifically designed for exploiting the structures of data and their potential fine-grained parallelism.

B. Parallel Execution of Data Processing

There has been a lot of work to exploit intra-query parallelism in parallel database systems. Pipelined parallel execution [4], [8], [19], [29], [37], [46], in which operators work in series by streaming their output to the input of the next one, is widely applied in database systems to exploit pipelined (vertical) parallelism. This approach was also extensively researched, especially in hash join algorithms in parallel database systems. These systems typically exhibit a small amount of parallelism and use a fixed degree of parallelism, which are usually determined by the number of operators of a query. Partitioned parallel execution [1], [7]–[9], [26], [27], in which input data is logically or physically partitioned into one or more nodes and operators are split into many independent ones working on the part of data, is also widely applied in parallel database systems to exploit partitioned (horizontal) parallelism. The amount of parallelism in these systems is determined by the number of partitions, which is also usually statically defined. ReDe employs a finer-grained dynamic parallel execution method, and we focused on introducing a way to apply the method to data lakes.

C. Fine-grained Parallel Execution of Data Processing

Recent work uses finer-grained task decomposition and distribution to fully take advantage of modern hardware capabilities. Morsel-driven query execution [28] takes small fragments of input data (morsels) and schedules these to worker threads that run entire operator pipelines at run-time to fully exploit the parallelism of many-core architecture. Out-of-order database execution [17] and scalable massively parallel execution [43] dynamically decompose query work during query execution to fully exploit the parallelism of underlying storage devices. The execution model of ReDe inherits the philosophy of the above work, but we focused on introducing a way to apply the above work to data lakes.

D. Modern Query Engines for Data Lakes

Modern query engines for data lakes focus on exploiting CPUs efficiently and follow one of two designs: either an interpreted vectorized design like in Photon [3] or a code-generated design like in HyPer [33] and Apache Impala [13]. ReDe takes a drastically different approach from these two designs to focus on optimizing I/O accesses by exploiting structures in data lake systems. This approach could be complementary to the approaches of such modern query engines; however, exploring how we can seamlessly integrate both approaches is our future work.

VII. Conclusion

We presented our vision for a new data management paradigm called *LakeHarbor*, which makes structures (e.g., indexes) first-class citizens in data lakes. LakeHarbor is yet another balanced solution for achieving the flexibility of data lakes and the performance of data warehouses with a special focus on structured data processing that fully exploits the parallelism of data. We also explored the potential of LakeHarbor by presenting ReDe, a prototype data processing engine that efficiently supports LakeHarbor, and a motivating evaluation and a case study of ReDe. We believe that the findings explored in this paper are beneficial for creating next-generation data processing systems on data lakes.

Acknowledgment

This work has been partially supported by FIRST and ImPACT research funding programs of Cabinet Office, Japan, Cross-cutting Technology Development for IoT Promotion program of NEDO and Big Data Value Co-creation Platform Engineering social cooperation program at UTokyo-IIS with Hitachi, and Cross-ministerial Strategic Innovation Promotion Program (SIP) on Integrated Health Care System. We are also grateful to Naohiro Mitsutake and Hiromasa Yoshimoto, who helped us conduct the case study appropriately.

References

[1] MC. Albutiu, A. Kemper, and T. Neumann. Massively Parallel Sort-merge Joins in Main Memory Multi-core Database Systems. *PVLDB*, 5(10):1064–1075, June 2012.

[2] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, J. Torres, H van Hovell, A. Ionescu, A. Łuszczak, M. Świtakowski, M. Szafrański, X. Li, T. Ueshin, M. Mokhtar, P. Boncz, A. Ghodsi, S. Paranjpye, P. Senster, R. Xin, and M. Zaharia. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *PVLDB*, 13(12):3411–3424, 2020.

[3] A. Behm, S. Palkar, U. Agarwal, T. Armstrong, D. Cashman, A. Dave, T. Greenstein, S. Hovsepian, R. Johnson, A. Sai Krishnan, P. Leventis, A. Luszczak, P. Menon, M. Mokhtar, G. Pang, S. Paranjpye, G. Rahn, B. Samwel, T. van Bussel, H. van Hovell, M. Xue, R. Xin, and M. Zaharia. Photon: A Fast Query Engine for Lakehouse Systems. In *SIGMOD*, page 2326–2339, 2022.

[4] MS Chen, ML Lo, PS. Yu, and HC. Young. Using Segmented Right-Deep Trees for the Execution of Pipelined Hash Joins. In *VLDB*, pages 15–26, 1992.

[5] Transaction Processing Performance Council. TPC-H is a Decision Support Benchmark. http://www.tpc.org/tpch/, 2023.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–149, 2004.

[7] D. DeWitt. DIRECT - a Multiprocessor Organization for Supporting Relational Data Base Management Systems. In *ISCA*, pages 182–189, 1978.

[8] D. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Commun. ACM*, 35(6):85–98, June 1992.

[9] D.J. DeWitt, J.F. Naughton, and J. Burger. Nested Loops Revisited. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 230–242, 1993.

[10] J. Dittrich, J. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a Yellow Elephant Run like a Cheetah (without It Even Noticing). *PVLDB*, 3(1–2):515–529, 2010.

[11] Apache Software Foundation. Apache Hadoop. https://hadoop.apache.org/, 2024.

[12] Apache Software Foundation. Apache Iceberg. https://iceberg.apache.org/, 2024.

[13] Apache Software Foundation. Apache Impala. https://impala.apache.org/, 2024.

[14] Apache Software Foundation. Apache Parquet. https://parquet.apache.org/, 2024.

[15] Apache Software Foundation. Apache Spark. https://spark.apache.org/, 2024.

[16] HL7 FHIR Foundation. HL7 FHIR Foundation Enabling health interoperability through FHIR. https://fhir.org/, 2024.

[17] K. Goda, Y. Hayamizu, H. Yamada, and M. Kitsuregawa. Out-of-Order Execution of Database Queries. *PVLDB*, 13(12):3489–3501, 2020.

[18] K. Goda and M. Kitsuregawa. *Powerful Analytics Platform for National-Scale Database of Health Care Insurance Claims*, pages 29–31. Springer Nature Singapore, 2022.

[19] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. QPipe: A Simultaneously Pipelined Relational Query Engine. In *SIGMOD*, pages 383–394, 2005.

[20] H. Hashimoto, M. Saito, J. Sato, K. Goda, N. Mitsutake, M. Kitsuregawa, R. Nagai, and S. Hatakeyama. Indications and classes of outpatient antibiotic prescriptions in Japan: A descriptive study using the national database of electronic health insurance claims, 2012–2015. *International Journal of Infectious Diseases*, 91:1–8, 2020.

[21] K. Hirayama, N. Kanda, H. Hashimoto, H. Yoshimoto, K. Goda, N. Mitsutake, and S. Hatakeyama. The five-year trends in antibiotic prescription by dentists and antibiotic prophylaxis for tooth extraction: a region-wide claims study in japan. *Journal of Infection and Chemotherapy*, 29(10):965–970, 2023.

[22] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *EuroSys*, pages 59–72, 2007.

[23] T. Ishikawa, J. Sato, J. Hattori, K. Goda, M. Kitsuregawa, and N. Mitsutake. The association between telehealth utilization and policy responses on COVID-19 in japan: Interrupted time-series analysis. *Interact. J. Med. Res.*, 11(2):e39181, 2022.

[24] T. Ishikawa, J. Sato, J. Hattori, K. Goda, M. Kitsuregawa, and N. Mitsutake. Changes in demand volume and patient/health care provider characteristics of first-time telehealth users: A comparative analysis before and after the COVID-19 policy response using the administrative claims database. *Telemed. J. E. Health.*, August 2023.

[25] N. Kanda, H. Hashimoto, T. Imai, H. Yoshimoto, K. Goda, N. Mitsutake, and S. Hatakeyama. Indirect impact of the covid-19 pandemic on the incidence of non-covid-19 infectious diseases: a region-wide, patient-based database study in japan. *Public Health*, 2022.

[26] C. Kim, T. Kaldewey, VW. Lee, E. Sedlar, AD. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-core CPUs. *PVLDB*, 2(2):1378–1389, August 2009.

[27] M. Kitsuregawa, H. Tanaka, and T. Moto-oka. Application of Hash to Database Machine and Its Architecture. 1(1):63–74, 1983.

[28] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age. In *SIGMOD*, pages 743–754, 2014.

[29] B. Liu and EA. Rundensteiner. Revisiting Pipelined Parallelism in Multi-Join Query Processing. In *VLDB*, pages 829–840, 2005.

[30] S. Madden, J. Ding, T. Kraska, S. Sudhir, D. Cohen, T. Mattson, and N. Tatbul. Self-organizing data containers. In *CIDR*, 2022.

[31] DG. Murray and S. Hand. Scripting the Cloud with Skywriting. In *HotCloud*, 2010.

[32] DG. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. CIEL: A Universal Execution Engine for Distributed Data-Flow Computing. In *NSDI*, page 113–126, 2011.

[33] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.

[34] R. Potharaju, T. Kim, E. Song, W. Wu, L. Novik, A. Dave, A. Fogarty, P. Pirzadeh, V. Acharya, G. Dhody, J. Li, S. Ramanujam, N. Bruno, C. Galindo-Legaria, V. Narasayya, S. Chaudhuri, A. Nori, T. Talius, and R. Ramakrishnan. Hyperspace: The Indexing Subsystem of Azure Synapse. 14(12), 2021.

[35] J. Sato, N. Mitsutake, M. Kitsuregawa, T. Ishikawa, and K. Goda. Predicting demand for long-term care using japanese healthcare insurance claims data. *Environ. Health Prev. Med.*, 27(0):42, 2022.

[36] J. Sato, N. Mitsutake, H. Yamada, M. Kitsuregawa, and K. Goda. Virtual patient identifier (vPID): Improving patient traceability using anonymized identifiers in Japanese healthcare insurance claims database. *Heliyon*, 9(5):e16209, 2023.

[37] Donovan A. Schneider and David J. DeWitt. Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines. In *VLDB*, pages 469–480, 1990.

[38] D. Taniar and J Rahayu. A Taxonomy of Indexing Schemes for Parallel Database Systems. *Distributed and Parallel Databases*, 12(1):73–106, 2002.

[39] R. Tsunoda, N. Mitsutake, T. Ishikawa, J. Sato, K. Goda, N. Nakashima, M. Kitsuregawa, and K. Yamagata. Monthly trends and seasonality of hemodialysis treatment and outcomes of newly initiated patients from the national database (NDB) of Japan. *Clinical and Experimental Nephrology*, 26(7):669–677, July 2022.

[40] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *SIGMOD*, page 1041–1052, 2017.

[41] M. Vuppalapati, J. Miron, R. Agarwal, D. Truong, A. Motivala, and T. Cruanes. Building An Elastic Query Engine on Disaggregated Storage. In *NSDI*, pages 449–462, 2020.

[42] T. Waki, K. Miura, S. Mizuno-Tanaka, Y. Ohya, K. Node, H. Itoh, H. Rakugi, J. Sato, K. Goda, M. Kitsuregawa, T. Ishikawa, and N. Mitsutake. Prevalence of Hypertensive Diseases and Treated Hypertensive Patients in Japan: A Nationwide Administrative Claims Database Study. *Hypertension Research*, pages 1123–1133, 2022.

[43] H. Yamada, K. Goda, and M. Kitsuregawa. Nested Loops Revisited Again. In *ICDE*, pages 3708–3717, 2023.

[44] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, MJ. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *NSDI*, pages 15–28, 2012.

[45] M. Zaharia, A. Ghodsi, R. Xin, and M. Armbrust. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *CIDR*, 2021.

[46] M. Ziane, M. Zaït, and P. Borla-Salamet. Parallel Query Processing with Zigzag Trees. *VLDBJ*, 2(3):277–302, July 1993.