Towards Unified Ad-hoc Data Processing

Xiaogang Shi[♯] Bin Cui[♯] Gillian Dobbie[§] Beng Chin Ooi[†]
[♯]Key Lab of High Confidence Software Technologies (MOE), School of EECS, Peking University
[§]Department of Computer Science, University of Auckland
[†]School of Computing, National University of Singapore
[♯]{sxg, bin.cui}@pku.edu.cn, [§]gill@cs.auckland.ac.nz, [†]ooibc@comp.nus.edu.sg

ABSTRACT

It is important to provide efficient execution for ad-hoc data processing programs. In contrast to constructing complex declarative queries, many users prefer to write their programs using procedural code with simple queries. As many users are not expert programmers, their programs usually exhibit poor performance in practice and it is a challenge to automatically optimize these programs and efficiently execute the programs.

In this paper, we present UniAD, a system designed to simplify the programming of data processing tasks and provide efficient execution for user programs. We propose a novel intermediate representation named UniQL which utilizes HOQs to describe the operations performed in programs. By combining both procedural and declarative logics, we can perform various optimizations across the boundary between procedural and declarative codes. We describe optimizations and conduct extensive empirical studies using UniAD. The experimental results on four benchmarks demonstrate that our techniques can significantly improve the performance of a wide range of data processing programs.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query Processing*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program Analysis*

Keywords

ad-hoc data processing, unified optimization, program analysis

1. INTRODUCTION

Ad-hoc data processing has proven to be a critical component for a variety of applications such as business intelligence, data mining and scientific computing. In a typical scenario, a user collects a set of data and has a list of questions about the data. As many ideas are hit on by accident, the lack of standard tools forces the user to write customized program himself to answer his questions.

Since many users are non-experts in programming, a simple but efficient method is needed for them to describe their tasks. Declarative languages are attractive tools as they can abstract away implementation details. Experienced users can write their programs with little effort and let optimizers choose efficient execution plans. Many existing data processing systems, such as relational databases, Pig [22] and Hive [28], provide declarative languages to users. But to achieve the benefits brought by these systems, programmers have to formulate their problems to specialized paradigms provided by the systems. These paradigms are usually limited in expressiveness and functionality. As many problems targeted by ad-hoc data processing are very complex, problem formulation in these systems is not a trivial task [35, 11]. Very often, users feel more comfortable writing their programs in procedural languages with embedded declarative primitives.

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2610492.

```
1  LOOP @d_id FROM 0 TO N
2      SELECT o_id INTO @o_id_list[@d_id]
3      FROM order
4      WHERE d_id = @d_id;
```

Figure 1: An example for optimizing

Though many systems offer general purpose programming languages integrated with declarative queries such as PL/SQL and LINQ [34], we observe that the optimization and execution for the procedural and declarative parts are still separated in these systems. For example, many modern databases allow users to write stored procedures with procedural extensions to SQL. But the procedural and declarative parts are executed in different execution engines. The procedural execution engine treats SQL queries as black boxes and calls database interfaces to execute the queries.

This separation makes it difficult to optimize programs. Consider the code fragment in Figure 1. If there is an index built on the attribute d_id, it performs well when executed. But if the index does not exist, unfortunately the executor will have to sequentially scan the entire table in each iteration of the loop. The performance would be significantly degraded due to the redundant table scans. Neither the program compiler nor the query optimizer can perform any optimization on the program — the program compiler does not have the knowledge of indexes while the query optimizer is unaware of the existence of the loop.

The performance is further hampered in the context of parallel processing due to the separate execution of procedural and declarative code. With the exponential growth in data size in many applications, there is an increasing need to process data in parallel. Although declarative queries can be easily executed in parallel, procedural code is usually executed sequentially. The benefit brought about by the parallelism of declarative queries is significantly limited by the interaction between the procedural code and the declarative queries. After submitting a query, the program has to wait until the results are returned. Then the program consumes the results by iterating over the results in sequence and performing computation

on the results. It becomes a bottleneck if the amount of data is big or the application logic is complex.

To improve the performance of these data processing programs, programmers have to examine both the procedural code and declarative queries in programs manually. Existing techniques in program compilation and query optimization cannot be directly applied to these programs. Attempts were made to allow automatic optimizations of such programs. D. Lieuwen et al. developed a rule-based optimizer to transform loops into joins [19] while R. Guravannavar et al. proposed to automatically rewrite iteratively invoked queries [13]. Some other studies focus on identifying the procedural code which can be translated into SQL queries [32, 33, 4]. The idea behind these attempts is to extract relational conditions from procedural code and construct equivalent SQL queries. Program performance might benefit from the query optimization in database systems. But as the relational algebra is unable to match the expressiveness of procedural languages, much procedural code cannot be translated into SQL queries and thus the benefits they gain are limited.

In this paper, we present a new system targeted for ad-hoc data processing, called UniAD, which stands for Unified execution for Ad-hoc Data processing. UniAD simplifies the programming of data processing tasks. Rather than constructing complex queries for their problems, users can write their programs in a procedural language with simple queries. User programs can be automatically optimized by UniAD and achieve good performance even if they are written poorly.

Unlike existing systems, UniAD takes both the procedural and declarative logics into consideration and can perform optimizations across the boundary between procedural code and SQL queries. These optimizations were hard to perform before due to the lack of a uniform representation suitable for the optimization of data processing programs. UniAD addresses the problem by translating user programs into a novel intermediate representation called Unified Query Language (UniQL).

UniQL deploys a simple and expressive mechanism, named High Order Query (HOQ), to describe used persistent data and performed operations in programs. HOQs provide a high level description of how persistent data is processed. SQL queries in programs can be easily translated into HOQs. Benefits gained from query optimization, including the utility of indexes, can be carried out in UniAD. Moreover, realizing that HOQs are logically equivalent to loops in procedural code, we can apply loop optimization techniques to HOQs. As operations described by SQL queries and procedural code are unified in HOQ, UniAD can find optimization opportunities previously ignored because of the separate optimization of procedural and declarative code.

UniAD also enables parallel execution of user programs even when programs are written assuming the sequential execution model. By combining computation logic with corresponding data in HOQs, UniAD can execute the computation logic in parallel in the place where its data resides.

To summarize, the main contributions of our work include:

- We design a new architecture to unify the optimization and execution of user programs, which are written in procedural languages with embedded declarative queries.
- We propose a novel intermediate representation that provides a uniform mechanism to describe data processing tasks in programs.
- We propose a transformation-based optimizer to automatically optimize programs. We show that the proposed intermediate representation allows concise and efficient implementation of many optimizations.
- We implement a prototype system and conduct extensive empirical studies on four different benchmarks including TPC-C [29] and SEATS [27]. We validate that we can achieve significant speed-ups for a variety of data processing programs.

The rest of the paper is organized as follows. We first introduce UniAD's intermediate representation in Section 2. Then we give an overview of UniAD and describe how UniAD executes user programs in Section 3. Details of program translation and optimization are presented in Section 4 and Section 5 respectively. Some practical issues in the implementation are discussed in Section 6. We evaluate our system in Section 7 and discuss related work in Section 8. Finally, we conclude the paper in Section 9.

2. HIGHER ORDER QUERY

It's challenging to optimize programs written in procedural languages with embedded SQL queries. Procedural code describes the exact operations performed in a program, whereas SQL queries provide a high level execution semantics and encapsulate detailed implementation methods. The encapsulation enables query optimizers to devise optimal execution plans for SQL queries. But meanwhile, the encapsulation creates a boundary between SQL queries and procedural code. This boundary makes it challenging to provide unified optimization and execution for procedural programs with embedded SQL queries. The problem is another form of the well-known *impedance mismatch* problem [20].

To address the problem, a uniform representation is needed to describe the operations in programs. The representation must satisfy the following requirements:

- First, the representation must be expressive and bridge the gap between procedural code and SQL queries. Relational algebra is beautiful yet powerful in representing queries on relations, but it is not capable of describing operations performed in procedural code such as loops and branches.
- Second, the representation must provide enough information for optimization and execution at a high level. Procedural code and other low level languages can express operations in SQL queries. However, because they eliminate the execution semantics, it is difficult to optimize the programs.
- Third, transformations on the representation must be available. The availability of transformations allows opportunities for program optimization.
- Finally, the representation must be as simple as possible. Unlike the operators provided to users, the representation is used for program optimization and execution in the system. A simple representation can reduce the complexity of optimization as the optimizer can further simplify various cases.

Some representations were proposed to solve the mismatch between declarative queries and procedural code [30, 1, 10], but they are not suitable for optimizing data processing programs. These representations have their roots in functional languages. Side effects such as I/O and shared states are eliminated, which makes it difficult to optimize the procedural code where side effects are frequently used.

Data-centric programming models have received considerable attention over the past few years. As one of the most well-known data-centric programming models, MapReduce was proposed by Google to simplify parallel data processing. A programmer can easily build their applications by specifying two relatively simple basic functions: *map* and *reduce*. Despite the simplicity of the paradigm, many useful computations can be efficiently abstracted.

```
1 /* selector */
2 ACCESS order WHERE o_w_id = @w_id
3 {
4   /* processor */
5   @total_amount += o_amount;
6   if(o_status == 'ready')
7     o_delivery_d = current_time;
8 }
```

Figure 2: An example of HOQ

These data-centric programming models share common characteristics in the manner in which they attach computation to corresponding data. Although these models resemble higher order functions in functional languages [18], user-defined functions are written in procedural languages. It provides a simple but expressive method to describe operations in the programs. By partitioning the data, application computation can also be executed simultaneously in the place where the data resides.

Inspired by the simplicity and expressiveness provided by data-centric programming models, UniAD uses HOQs to describe operations in the programs. An example of HOQ is given in Figure 2. Each HOQ consists of a selector and a processor. The selector uses a WHERE clause to indicate the tuples selected. The statements below the selector comprise the HOQ's processor, which describe the operations performed on each selected tuple. When viewed in this form, a processor acts as an anonymous function and is bounded to each selected tuple in the HOQ. So a HOQ can be viewed as a higher-order version of a SQL query, hence its name.

Algorithm 1 HOQ Execution Semantics

```
1: for each tuple t in table do
2:     if t satisfies HOQ's selector then
3:         perform HOQ's processor on t
4:     end if
5: end for
```

HOQ's execution semantics is illustrated in Algorithm 1 and is easy to understand. Once a qualified tuple is selected, the operations defined in the HOQ's processor are performed on the tuple.

Using the HOQs, UniQL is an efficient intermediate representation for programs in UniAD. First, by breaking the encapsulation of SQL queries and translating SQL queries into HOQs, we can have a clear and uniform picture about the data used in the programs and operations performed on the data. Such information can be used to guide our optimizations.

Second, HOQ is capable of describing a wide variety of execution plans. It allows us to find efficient execution plans which cannot be expressed in relational algebra. Consider the operations performed in Figure 2. The code fragment calculates the sum of field o_amount and for those tuples whose field o_status is *'ready'*, updates their field $o\_delivery\_d$ to current time. The HOQ in Figure 2 provides an efficient execution plan to perform these operations. These operations can be performed by scanning the table $order$ in one pass. Although many features, such as MERGE and window functions, have been added into the SQL standard to support efficient execution plans, it is still impossible to use relational algebra to express the query described in Figure 2.

Finally, HOQ can facilitate concise and efficient implementation of many program optimizations. As the selected data is described in a declarative manner, the executor is free to choose different methods to fetch data. With the knowledge of physical storage and available indexes, the executor can fetch data efficiently. Moreover, note that the HOQ's execution sequentially iterates over retrieved data, we can conceptually treat HOQs as loops. Hence, many well-studied loop optimization techniques can be applied to HOQs.

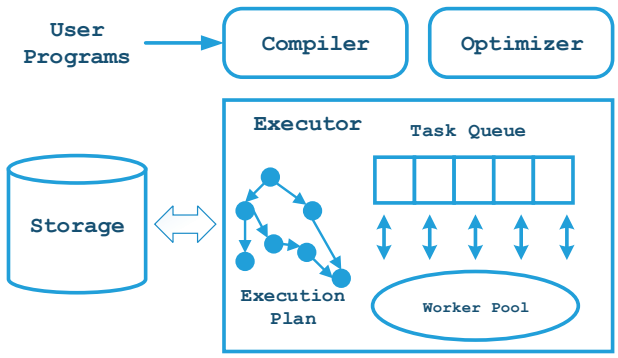


Figure 3: System architecture

3. SYSTEM OVERVIEW

In this section, we will introduce the UniAD system and describe how a program is compiled, optimized and executed in UniAD.

The high-level architecture of UniAD is illustrated in Figure 3. UniAD deploys a relational data model. User data is structured as a collection of tables and stored in the underlying storage. UniAD also allows users to create indexes if needed, and stores the information about tables and indexes in the catalog.

The language provided by UniAD is similar to those procedural languages in modern databases which are extensions to the SQL language. But UniAD differs from these databases which only provide execution for SQL queries, in that UniAD provides unified execution of user programs.

When a user program is passed to UniAD, it is parsed by the compiler. To enable further optimizations, the compiler translates the SQL queries in the program into HOQs. For example, the code fragment in Figure 1 is translated into the following UniQL code:

```
LOOP @d_id FROM 0 TO N
  ACCESS order WHERE d_id = @d_id
    @o_id_list[@d_id].append(o_id);
```

Details of program translation will be described in Section 4.

Program analysis is also applied to the translated program and the output of the compilation is a directed cyclic graph. An example of the graph is illustrated in Figure 4. Nodes in the graph are statements to be executed and edges show the dependency between these nodes. There are three types of nodes in the graph, namely basic blocks, HOQs and loops. They are represented by B, Q and L in the graph respectively. Branches are translated into conditions in the edges.

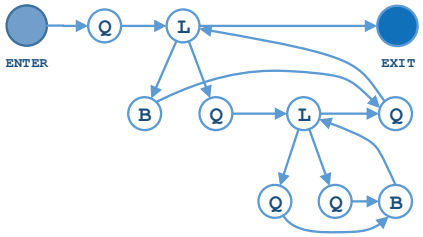


Figure 4: An execution plan graph

The graph represents an execution plan for the program. It is different from a control flow graph as we allow more than one node to be executed simultaneously. A node can not be executed until all nodes directed to it have completed their execution. Loop nodes are handled as a special case since they lead to cyclic dependencies in the graph. The nodes directing to a loop node can be divided into two categories according to whether they are inside the loop's body or not. If a node is outside the loop's body, the loop node cannot

be executed until the node finishes its execution. But once the loop node has started its execution, it will not depend on the node any more. It will be re-executed when all the nodes inside its loop body have finished their execution.

The optimizer in UniAD is invoked to find a better execution plan for the program. The optimizer consists of a set of transformation rules. UniAD deploys a greedy heuristic optimization algorithm, which examines the execution plan graph iteratively and performs a transformation on the graph if the transformation rule's condition is satisfied. The optimizer stops when it cannot find any further optimization opportunities. For example, realizing that the above code fragment has to scan the table repeatedly, the optimizer will interchange the HOQ and loop, and generate the following HOQ:

```
ACCESS order
  LOOP @d_id FROM 0 TO N
    IF(d_id == @d_id)
      @o_id_list[@d_id].append(o_id);
```

Since the HOQ can be executed by scanning the table once, the performance of the original code fragment is improved. We will introduce the transformation rules used in UniAD in Section 5.

After the execution plan for the program is generated, an executor responsible for the program's execution will be created. Each executor consists of a set of worker threads and operations in the execution plan will be executed by these workers. The executor maintains a task queue which contains all nodes that can be executed. At the beginning, only the program's enter node is put in the task queue. Each idle worker picks a task from the task queue and executes the task. To utilize the data parallelism, data is partitioned and assigned to different workers. HOQs will be decomposed into a set of instances and executed by corresponding workers.

When a task is completed, the worker examines all the nodes directly connected to the task. A node will be put in the task queue if all the nodes connected to it have completed their execution. The execution of the program is completed when the program's exit node is executed.

4. PROGRAM TRANSLATION

When a user program is passed to the compiler, SQL queries in the program are translated into HOQs. We mainly focus on SQL queries with standard SQL features such as selection, projection, join, aggregation, sorting and from clause subqueries. To preserve the efficiency brought by query optimization, UniAD uses an embedded query optimizer to generate an optimal execution plan for each SQL query in the program and translates queries into HOQs according to their execution plans.

The execution plan for an SQL query is usually expressed as a query plan tree (QPT). Each node in the QPT represents an operator needed to execute the query along with the method to implement the operator. Each SQL query can be translated into HOQs by traversing its QPT in post order and converting each visited operator according to the operator's implementation method.

```
SELECT SUM(l_extendedprice)/7.0 INTO @sum_price
FROM lineitem, part,
    (SELECT l_pkey AS aq_pkey,
            0.2*AVG(l_quan) AS aq_quan
    FROM lineitem
    GROUP BY l_pkey) AS avgquantity
WHERE p_brand = 'BRAND#1'
AND p_container = 'LG CASE'
AND p_pkey = l_pkey
AND p_pkey = aq_pkey
AND l_quan < aq_quan;
```

The above SQL query is a variation of TPC-H Q17 which contains rich SQL features, and we use it as an example to illustrate the translation. The QPT for the query is illustrated in Figure 5 while the generated code fragment is illustrated in Figure 6.

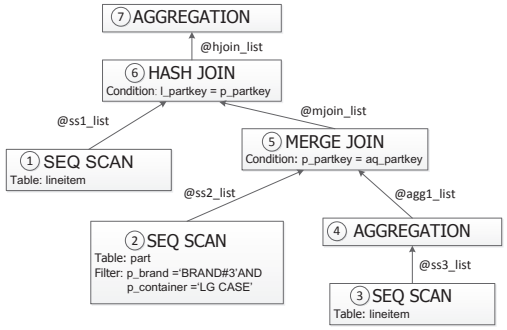


Figure 5: A query plan tree

```
1  /*NODE 1: Generated HOQ to scan the table*/
2  ACCESS lineitem
3    @ss1_list.append(MAKE_TUPLE(l_pkey, l_quan, ...));
4
5  /*NODE 2: Generated HOQ to scan the table*/
6  ACCESS part
7  WHERE p_brand = 'BRAND#1'
8    AND p_container = 'LG CASE' {
9    @ss2_list.append(p_pkey);
10 }
11
12 /*NODE 3: Generated HOQ to scan the table*/
13 ACCESS lineitem
14   @ss3_list.append(MAKE_TUPLE(l_pkey, l_quan));
15
16 /*NODE 4: Generated code to perform aggregation*/
17 @agg1_list = ... @ss3_list ... ;
18
19 /*NODE 5: Generated code to perform merge join*/
20 @mjoin_list = ...@ss2_list...@agg1_list...;
21
22 /*NODE 6: Generated code to perform hash join*/
23 @hjoin_list = ...@ss1_list...@mjoin_list...;
24
25 /*NODE 7: Generated code to perform aggregation*/
26 @sum_price = ...@hjoin_list...;
```

Figure 6: Generated code fragment

Each node in the QPT is labeled with the sequence number of the visit in post order iteration. Except for the root node, all other nodes in the QPT have a parent node. Each node's output is fed as input to the parent node. We format both input and output of each node as lists of tuples and use local variables to store these intermediate results. For example, we store the tuples selected by Node 1 into $@ss1\_list$, which is then used by Node 6 to produce $@hjoin\_list$.

There are three leaf nodes in the QPT and they are all scan nodes. A HOQ will be constructed for each of them by putting the scan node's predicate in the selector and creating necessary statements in the processor. Other nodes in the QPT are internal nodes and necessary procedural code will be generated to implement these nodes' functionality. UniAD maintains a set of code templates and each operator will be translated into procedural code according to the method selected by the query optimizer.

A code template for merge join operators is illustrated in Figure 7, which contains five parameters. #ListA and #ListB are two tuple lists to perform join; #ExprA and #ExprB are expressions in the join condition, and #OutputList is the result list. To translate the operator defined in Node 5, we will pass necessary arguments to the template and generate code.

The translation is in essence similar to query rewriting in databases. Our method resembles the ones proposed in [17, 21], but we focus on high-level execution semantics. We attempt to keep the translation process as simple as possible and do not take the data and instruction locality into consideration. The complexity of the

```
1  MergeJoin<#ListA, #ExprA, #ListB, #ExprB, #OutputList>
2    SORT #ListA ON #ExprA
3    SORT #ListB ON #ExprB
4    LOOP @i FROM 0 TO #ListA.size()
5      LOOP @j FROM 0 TO #ListB.size()
6        IF(#ExprA < #ExprB)
7          @i++;
8        ELSE IF(#ExprA > #ExprB)
9          @j++;
10       ELSE  {
11           #OutputList.append(MAKE_TUPLE(...));
12           @i++;
13           @j++;
14       }
```

Figure 7: A code template for merge join operator

translation is $O(N)$ where N is the number of relational operators in the program. As the number of relational operators is limited in a given program, the translation overhead is considered proportional to the code size. The code size is generally orders of magnitude less than the data size, and hence the translation overhead is negligible. Some intermediate variables and inefficient code will be introduced by the translation, but they will not degrade the overall performance as the optimizer can eliminate this inefficient code at a later stage in our system.

5. OPTIMIZATION

In this section, we present the details of the optimizations performed in UniAD. We focus on those optimizations across the boundary between the procedural and declarative parts. These optimizations share similar ideas to some techniques in program compilation and query optimization, which however cannot be directly applied to user programs in UniAD.

We begin with the introduction of our optimizations, using an example to illustrate the transformation and its benefits. We then discuss the prerequisites for the rules that are necessary to ensure that a program's results are consistent. To ensure the correctness, we reduce our transformations to other transformations that are well-studied in the program analysis field. By applying data dependency analysis [16] to programs, we check each transformation's validity before it is performed. Finally, a formal specification of the optimization rule and required preconditions are given.

5.1 Preliminary Notation

Before we introduce our transformation rules, we shall first describe the symbols we use in the rules.

An ordered list $\mathcal{B} = [\mathtt{s}_1, \mathtt{s}_2, ..., \mathtt{s}_n]$ is used to express a list of statements in sequence. The operator $\odot$ is used to concatenate two statement lists. Suppose $\mathcal{B}_1 = [\mathtt{s}_1, \mathtt{s}_2, ..., \mathtt{s}_n], \mathcal{B}_2 = [\mathtt{t}_1, \mathtt{t}_2, ..., \mathtt{t}_m]$, then $\mathcal{B}_1 \odot \mathcal{B}_2 = [\mathtt{s}_1, \mathtt{s}_2, ..., \mathtt{s}_n, \mathtt{t}_1, \mathtt{t}_2, ..., \mathtt{t}_m]$.

$\mathtt{s}_{[a/b]}$ is used to express the statement produced by replacing all variables named a in statement $\mathtt{s}$ by b.

There are four types of statements in translated programs, namely assignments, loops, branches and HOQs.

Assignments are statements assigning expressions to variables. We use *Assign(*v, E*)* to express the statement of the form $v = \mathsf{E}$.

Branches are statements with conditions. We use *Branch(*C, $\mathcal{B}$*)* to express a statement of the form ***if*** C ***do*** $\mathcal{B}$, where C is the condition expression and $\mathcal{B}$ contains the statements to be executed when the condition holds.

Loops are statements executed repeatedly. We use *Loop(*I, $\mathcal{B}$*)* to express a loop of the form ***foreach*** i ***in*** I ***do*** $\mathcal{B}$, where I is the iterator space and $\mathcal{B}$ contains the statements executed in each iteration.

HOQs of the form ***access*** S ***do*** $\mathcal{P}$ are expressed by *HOQ(*S, $\mathcal{P}$*)* where S is the selector to filter tuples and $\mathcal{P}$ contains the statements in the HOQ's processor.

5.2 Extracting Conditions

Predicates in a HOQ's selector indicate the tuples to be processed by the HOQ. If only on a subset of selected tuples are processed by the HOQ, it is, of course, inefficient to fetch unused tuples.

Consider the following code fragment:

```
ACCESS order
  IF(w_id == 30)
    @cost += price;
```

As no predicate is given in the HOQ's selector, the selector has to scan the entire table sequentially and select all tuples in the table. Note that the statement '$@cost+ = price$' is executed only when the accessed tuple's attribute w_id is 30. If there is an index built on the attribute w_id, redundant tuple accesses can be eliminated by fetching tuples from the index with the condition in the branch statement. Therefore, the following HOQ is more efficient than the above one:

```
ACCESS order WHERE w_id = 30
    @cost += price;
```

The condition '$w\_id == 30$' is now put in the selector and it is extracted from the common condition of branch statements in the processor. The transformation is analogous to *predicate pushdown* in query optimization while viewed in the program optimization, it is another form of *loop-invariant code motion*. To preserve the semantics, the extracted common condition should be loop-invariant in the HOQ.

The transformation rule can be formalized as follows:

RULE 1 (EXTRACT CONDITIONS). *Suppose* q *= HOQ(*S*,* $\mathcal{P}$*) is a HOQ,* $\mathcal{P} = \left[Branch(\mathsf{C}_i, \mathcal{B}_i)\right]$ *and* $\mathsf{CC} = \bigcup \mathsf{C}_i$*. q can be transformed to* q' *= HOQ(*S'*,* $\mathcal{P}'$*), where* $\mathsf{S}' = \mathsf{S} \cap \mathsf{CC}$ *and* $\mathcal{P}' = \left[Branch(\mathsf{C}_i, \mathcal{B}_i)\right]$ *if* CC *is loop-invariant in the HOQ.*

The condition of the branch statement can be eliminated if it's a subset of the condition in the selector.

5.3 Fusing HOQs and Loops

Many programs use sequential loops to examine the tuples produced by declarative queries and perform computation on these tuples one tuple at a time. We can combine the computation performed in the loop with the HOQ and replace the loop and the HOQ with a new HOQ.

The transformation is a very useful optimization in UniAD. In combination with *dead code elimination*, the transformation can help us remove useless intermediate variables which are introduced in the translation. Another benefit brought about by the transformation is that by attaching the computation to selected tuples, it is possible for us to perform more computation in situ and in parallel. Besides, the transformation binds the computation with the corresponding data and shields the processing details performed on the data. As we will see in the later section, the transformation can help us focus on the selected data and find more opportunities for optimization.

As each HOQ can be viewed as a loop, the transformation is essentially loop fusion or function composition. It is not always safe to fuse a HOQ and a loop, dependencies in both the HOQ and loop have to be examined. Note that the order in which a HOQ is iterated is not preserved, we enhance the requirement by prohibiting any loop fusion which will lead to loop-carried dependencies in the combined HOQ. With such a requirement, no loop-carried dependencies will exist in the HOQ then the HOQ can be executed in parallel safely.

The requirement that no loop-carried dependencies are allowed in the HOQ is too rigid and as a consequence we will miss many opportunities for transformation. The requirement can be relaxed if loop-carried dependencies are caused by commutative operators.

Commutative operators can produce identical results regardless of the order they are executed. We notice that commutative operators are common in programs, so it is necessary to provide parallel execution for these operators. UniAD predefines a set of commutative operators, such as append and addequal. We can execute these operators in parallel to get partial results and merge partial results to get a final result.

The transformation rule can be formally specified as follows:

RULE 2 (FUSE HOQS AND LOOPS). *Suppose* q = *HOQ*(S, $\mathcal{P}$) *is a HOQ,* l = *Loop*(I, $\mathcal{B}$) *is a loop.* $\left[\ q, l\ \right]$ *can be transformed to* $\left[q'\right]$ = $\left[\ HOQ(\mathsf{S}, \mathcal{P}')\ \right]$, *where* $\mathcal{P}' = \mathcal{P} \odot \mathcal{B}$, *if the following conditions are satisfied:*

- *There exists a one-to-one producer-consumer relationship between* q *and* l.
- *All dependencies in* q' *are caused by commutative operators.*

A special case of the transformation is that the output of a HOQ is only one tuple. In such a case, we can also put all statements consuming the output into the HOQ.

5.4 Merging HOQs

Multiple Query Optimization (MQO) [26] is an efficient method to improve program performance by sharing common intermediate results and eliminating redundant tuple accesses. However, we observe that opportunities to perform MQO are often limited in practice.

Consider the following code fragment. It is selected from a transaction in TPC-C named $NewOrder$.

```
SELECT s_quantity INTO @s_quantity
FROM Stock
WHERE s_i_id = @ol_i_id
AND s_w_id = @ol_supply_w_id;

IF(@s_quantity > @ol_quantity)
  @s_quantity = @s_quantity - @ol_quantity;
ELSE
  @s_quantity = @s_quantity - @ol_quantity + 91;

UPDATE Stock
SET s_quantity TO @s_quantity
WHERE s_i_id = @ol_i_id
AND s_w_id = @ol_supply_w_id;
```

The program first retrieves the item's quantity from table $stock$ and stores the value into $@s\_quantity$. After $@s\_quantity$'s value is computed in the branch statement, $@s\_quantity$'s value is written back to the tuple's field $s\_quantity$ by the update statement. Both the SELECT and UPDATE statements have to probe the table to fetch selected tuples. Given that they actually access the same tuples, it's inefficient to select the tuples twice. MQO can't be directly used to merge these queries and reduce the redundant table access because MQO cannot deal with the procedural code.

By translating SQL queries into HOQs, it is easier for us to merge the queries. We find that the branch statement can be fused with the first HOQ as it is a consumer of the HOQ. After applying Rule 2, we get the following code fragment:

```
ACCESS Stock
WHERE s_i_id = @ol_i_id
AND s_w_id = @ol_supply_w_id {
  @s_quantity = s_quantity;
  IF(@s_quantity > @ol_quantity)
    @s_quantity = @s_quantity - @ol_quantity;
  ELSE
    @s_quantity = @s_quantity - @ol_quantity + 91;
}
ACCESS Stock
WHERE s_i_id = @ol_i_id
AND s_w_id = @ol_supply_w_id {
  s_quantity = @s_quantity;
}
```

As the two HOQs have the same selector, it is clear that the two HOQs operate on the same set of tuples. We can combine them by simply putting statements in their processors together. We get the following code fragment:

```
ACCESS Stock
WHERE s_i_id = @ol_i_id
AND s_w_id = @ol_supply_w_id {
  @s_quantity = s_quantity;
  IF(@s_quantity > @ol_quantity)
    @s_quantity = @s_quantity - @ol_quantity;
  ELSE
    @s_quantity = @s_quantity - @ol_quantity + 91;
  s_quantity = @s_quantity;
}
```

Compared with the original code fragment which has to fetch the tuples twice, the above HOQ can be executed by fetching the tuples only once.

We can generalize the transformation by allowing the merging of any two HOQs that access a common subset of tuples. Branches should be added to make the combined HOQ perform properly. Given that merging two HOQs is another form of loop fusion, examinations needed for loop fusion are also required here. The transformation rule can be formulated as follows:

RULE 3 (MERGE HOQS). *Given two HOQs* q_1 = *HOQ*(S_1, $\mathcal{P}_1$) *and* q_2 = *HOQ*(S_2, $\mathcal{P}_2$), $\left[\ q_1, q_2\ \right]$ *can be transformed to* $\left[q = HOQ(\mathsf{S}, \mathcal{P})\right]$ *where* $\mathsf{S} = \mathsf{S}_1 \cup \mathsf{S}_2$ *and* P = $\left[\mathit{Branch}(\mathsf{S}_1, \mathcal{P}_1), \mathit{Branch}(\mathsf{S}_2, \mathcal{P}_2)\ \right]$ *if the following conditions are satisfied:*

- S *is loop-invariant in both* q_1 *and* q_2.
- *All loop-carried dependencies in* q *are caused by commutative operators.*

5.5 Prefetching Results

Prefetching is an efficient method to reduce program latency. UniAD's optimizer utilizes prefetching to reduce unnecessary waits in programs when two HOQs can be executed simultaneously.

Consider the following code fragment:

```
Q1: ACCESS customer WHERE c_name = @c_name
      @c_id = c_id;
    IF(@c_id == 0)
Q2:   ACCESS frequentflyer WHERE ff_name = @c_name
        @c_id = ff_c_id;
```

The code fragment retrieves a customer's id by its name. It first probes the table $customer$ to find the customer's id. If the customer's id is not found in the table $customer$, it then proceeds to probe the table $frequentflyer$.

In the above code fragment, Q2 cannot be executed until Q1's result is returned. When the condition in the branch statement is satisfied, we have to scan table $customer$ and $frequentflyer$ in sequence. The program latency is $t_1 + t_2$, where t_1 and t_2 are the time to execute Q1 and Q2 respectively.

As the execution of Q2 does not depend on Q1's result, we can execute Q2 speculatively and use an variable to prefetch Q2's result. If the condition in the branch statement is satisfied, we will use the prefetched result, without probing the table $frequentflyer$ again.

The following is the transformed code fragment:

```
ACCESS customer WHERE c_name = @c_name
  @c_id = c_id;
ACCESS frequentflyer WHERE ff_name = @c_name
  @prefetch_c_id = ff_c_id;
IF(@c_id == 0)
  @c_id = @prefetch_c_id;
```

Since the two HOQs can be executed simultaneously, the program latency of the above code fragment is $\max(t_1, t_2)$, which is less than the original one.

We use $def(\mathcal{B})$ to refer to the variables defined in the statement list $\mathcal{B}$. Then the formal transformation rule is given as follows:

RULE 4 (PREFETCH RESULT). *Given two HOQs* q_1 = *HOQ*(S_1, $\mathcal{P}_1$) *and* q_2 = *HOQ*(S_2, $\mathcal{P}_2$), *a branch* b = *Branch*(C, $[\ q_2\ ]$). $[q_1, b]$ *can be transformed to* $[q_1] \odot \mathcal{B}_1 \odot [q_2'] \odot \mathcal{B}_2$, *where* $\mathcal{B}_1$ = $[Assign(v', v)]$, q_2' = *HOQ*(S_2, $[\ \mathsf{s}_{[v/v']} \mid \mathsf{s} \in \mathcal{P}_2 \wedge v \in def(\mathcal{P}_2)\ ]$), $\mathcal{B}_2$ = $[$ *Branch* $(\ \mathsf{C}, [$ *Assign*(v, v') $\mid v \in def(\mathcal{P}_2)\ ]\ )\ ]$ *if* q_2 *can be executed simultaneously with* q_1.

The transformation may lead to unnecessary execution when the branch statement's condition is not satisfied. In practice, we only choose to apply the transformation when the speculatively executed HOQ can be merged with previous HOQs.

5.6 Interchanging Loop and HOQ

Recall the code fragment in Figure 1. By translating the SQL query into a HOQ, we get the following code fragment:

```
LOOP @d_id FROM 0 TO N
  ACCESS order WHERE d_id = @d_id
    @order_list[@d_id].append(o_id);
```

If there is no index built on the attribute d_id, the executor will sequentially scan the table at each iteration of the loop. The redundant table scans will degrade the performance significantly. Given that the HOQ is logically also a loop, we can interchange the HOQ and the loop and get the following code fragment:

```
ACCESS order
  LOOP @d_id FROM 0 TO N
    IF(d_id == @d_id)
      @order_list[@d_id].append(o_id);
```

The loop now is nested in the HOQ. Though the complexity of the code fragment is unchanged, the above code fragment eliminates the redundant table scans and can be executed with only one table scan.

As in the case of program optimization, loop interchange may reorder the endpoints of a dependency, which will lead to an invalid transformation. We should check the dependencies in the loop and ensure the safety of the transformation.

The transformation can be specified as follows:

RULE 5 (INTERCHANGE HOQS AND LOOPS). *Suppose* l = *Loop*(I, $[q]$) *is a loop and* q = *HOQ*(S, $\mathcal{P}$) *is a HOQ.* l *can be transformed into* q' = *HOQ*($\emptyset$, $[l']$), *where* l' = *Loop*(I, *Branch*(S, $\mathcal{P}$)) *if the following conditions are satisfied:*

- *No indexes are available for* S.
- *Endpoints of all dependencies in* q *are not reordered in* l'.

5.7 Flattening Nested Loops

Though Rule 5 provides a transformation to optimize the code fragment in Figure 1, we can execute the code fragment more efficiently.

We begin with the code fragment in Figure 1. Due to the logical equivalence of HOQs and loops, we can express the HOQ nested in the loop in the loop format:

```
LOOP @d_id FROM 0 TO N
  FOREACH t IN order
    IF(t.d_id == @d_id)
S:    @order_list<@d_id>.append(t.o_id);
```

The value of field o_id is fetched in S if the branch statement's condition is satisfied. Using the idea described in Section 5.5, we can speculatively prefetch the field's value into a hash map and assign the value in the map to $@order\_list[@d\_id]$. Then we get:

```
LOOP @d_id FROM 0 TO N
  FOREACH t IN order
    @o_id_map<t.d_id>.append(t.o_id);
    IF(t.d_id == @d_id)
      @order_list[@d_id] = @o_id_map<@d_id>;
```

Note that we store the value in $@o\_id\_map < t.d\_id >$ but access the value by $@o\_id\_map < @d\_id >$. It is allowed as we only access the value when $t.d\_id == @d\_id$.

As there does not exist any cycle of dependencies in the loop on table $order$, we can split the loop and get the following code fragment:

```
    LOOP @d_id FROM 0 TO N
L1:   FOREACH t IN order
        @o_id_map<t.d_id>.append(t.o_id);
L2:   FOREACH t IN order
        IF(t.d_id == @d_id)
          @order_list[@d_id] = @o_id_map<@d_id>;
```

The loop on table $order$ now splits into two loops inside the outer loop, namely L1 and L2. The variables inside L1 are all invariant in the outer loop, hence we can put L1 outside the outer loop. The branch statement inside L2 is also invariant in L2, we can directly remove L2. Finally, we get the following code fragment:

```
ACCESS order
  @o_id_map<d_id>.append(o_id);
LOOP @d_id FROM 0 TO N
  @order_list[@d_id] = @o_id_map<@d_id>;
```

Both the original code fragment and the code fragment optimized by Rule 5 contain nested loops and their complexity is $O(N*M)$ if the table's size is M. If $@o\_id\_map$ is a hash map, both insert and search operations on the map are performed in $O(1)$ generally. The complexity of the code fragment above is $O(max(M, N))$, which is much better than both of its counterparts.

This transformation rule provides a more efficient optimization on programs than Rule 5, but it is limited as it can only deal with HOQs with equality predicates. It can be easily understood when explained in the context of joins. The original code fragment actually implements an *equi-join* between a temporary table $\{@d\_id\}$ and the table $order$ using *nested loop join*. The intuition behind the transformation is that we choose to implement the equi-join using *hash join* instead.

Both loop fission and loop-invariant code motion are used in the transformation. Their safety will be taken into consideration when the transformation is applied. To split the loop on table $order$, the statements in the HOQ should not contain any cycles of dependencies. Besides, both the variable $@order\_list[@d\_id]$ and the expression $@d\_id$ should also be invariant in the loop on table $order$. Otherwise, we cannot delete the loop L2.

The transformation can be described as follows:

RULE 6 (FLATTEN NESTED LOOPS). *Suppose* l = *Loop*(I, $[q]$) *is a loop and* q = *HOQ*(c=e, $\mathcal{P}$) *is a HOQ,* v *is a variable defined in* $\mathcal{P}$ *and* c *is a column .* $[l]$ *can be transformed into* $[\ q', l'\ ]$ *where* q' = *HOQ*($\emptyset$, $\{\mathsf{s}_{[v/map<c>]} | \mathsf{s} \in \mathcal{P}\}$) *and* l' = *Loop*(I, [*Assign*(v, *map*<e>)]), *if the following conditions are satisfied:*

- *No indexes are available for* $c = \mathsf{e}$.
- *There is no cycle of dependencies in* $\mathcal{P}$.
- v *and* e *are loop-invariant in* q.

6. IMPLEMENTATION DETAILS

In this section, we will discuss some practical issues in the system implementation and introduce our methods for addressing these issues.

6.1 Heap Synchronization

When a HOQ is executed in parallel, the HOQ will be sent to all corresponding worker threads and an instance of the HOQ will be executed in each worker thread.

In UniAD, each program's heap is shared among all worker threads executing the program. As a HOQ can access free variables that are defined outside their lexical scope, worker threads might access shared data simultaneously when executing the HOQ. Though data dependency analysis ensures that the operations performed on the data do not conflict with each other, contention is still possible in

practice. The contention may arise from the explicit updates on a loop iterator or implicitly when an element is inserted into a map.

One method to address the problem is to use a fine-grained locking mechanism to ensure strictly exclusive access to shared data. But this method is too costly and performance might be harmed. It also reduces the parallelism as threads might be queued when performing updates on shared data.

Another method to solve the problem, which is borrowed from many programming languages that support closures, is to create a referencing environment for each executing instance of the HOQ and copy all variables into the environment which the HOQ instance will use in the execution. By doing this, the execution of each instance is completely independent as their heaps are separated. This method requires an accurate analysis of the free variables used in the execution to avoid unnecessary data copies. It is not trivial as the data that an instance will access cannot be easily known before it is executed.

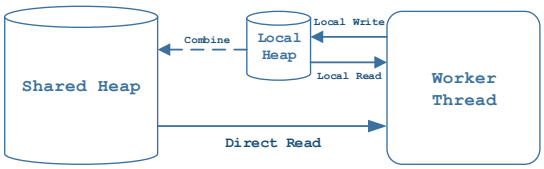


Figure 8: Heap synchronization

UniAD adopts the idea of contention detection proposed in [5], and modifies it to avoid contention between different threads in our implementation. The method is illustrated in Figure 8.

We create a local heap for each worker thread. When the worker thread needs to read a variable's value, it first probes the value in its local heap. If the variable does not reside in the local heap, then the worker proceeds to probe it from the shared heap. Recall that each instance of the same HOQ does not rely on the outputs of other instances, outputs of the worker thread will be buffered in its local heap. After all instances of the HOQ are completed, their outputs will be pulled out from local heaps and merged into the shared heap. Additional computation will be needed for those variables whose values are produced by commutative operators.

6.2 Consistent Execution

UniAD allows different programs executing concurrently by deploying the two-phase locking (2PL) protocol. Each program performs as a transaction in a database. Before accessing a tuple, the program must hold the locks associated with it and held locks are released when the program completes its execution. The locking protocol can solve the problem of synchronizing access to shared data, but inconsistent execution may still happen in UniAD.

Consider the following HOQ:

```
ACCESS item WHERE quantity < @threshold
  quantity -= @delta;
```

For each tuple whose $quantity$ is less than $@threshold$, its value of $quantity$ is subtracted by $@delta$. Suppose tuples are fetched using a B^+-Tree index built on the attribute $quantity$. As the tuple's entry in the index will also be updated when its value of $quantity$ is updated, a tuple might be revisited and be fetched more than once, which will lead to incorrect results.

Concurrency control mechanisms, including the locking protocol, do not help solve the problem as the problem is not caused by concurrent accesses of different programs. The problem is due to the inconsistent snapshots seen at each iteration of the HOQ. During the execution of a HOQ, the HOQ's processor will be invoked each time a qualified tuple is selected. Each qualified tuple should be selected only once in the execution. But the updates performed in the processor may violate the requirement.

The problem is similar to the problem caused by performing updates within a *sensitive cursor* in stored procedures. Most databases solve it by materializing the set of data used for the cursor and storing the entire set in a temporary table. It is not efficient if a large number of tuples are fetched.

UniAD solves the problem by storing multiple versions of the same tuple. In UniAD, each tuple has a version number TV. UniAD maintains a number CV to record the number of writes which have been processed. When an update is performed on a tuple, a new version of the tuple will be created. The value of CV is incremented and its value will be used as the new tuple's TV. When a HOQ is to be executed, it first gets the current value of CV and stores it into QV. Only those tuples whose TV is less than the HOQ's QV are visible to the HOQ. Since those tuples produced during the HOQ's execution must have a larger TV than the HOQ's QV, they will not be processed by the HOQ.

The mechanism is similar to a timestamp-based implementation of Multi-version Concurrency Control (MVCC), except in the standard implementation of MVCC, a new tuple produced by a HOQ would be visible to the program immediately. However in UniAD, the tuple cannot be seen by the program until the HOQ's execution is completed.

7. EVALUATION

In this section, we present the experimental evaluation of our UniAD system. The evaluation consists of two parts. We first evaluate the benefits brought by the optimization techniques in Section 7.1, followed by the capability of UniAD to execute the programs in parallel in Section 7.2. All the experiments are performed on a CentOS-5.5 server with 48GB RAM and two Intel E7-4807 processors, each of which contains six cores clocked at 1.86GHz.

7.1 Evaluation on Program Optimization

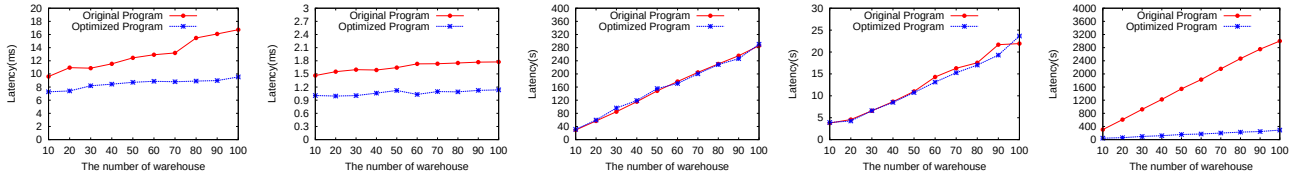
We first evaluate how effectively UniAD can optimize the user programs. The programs from four different benchmarks are adopted in the experiments: TPC-C [29], SEATS [27], PageRank and TF-IDF. They together represent a variety of programs for data processing, e.g., online transaction processing and data analytic applications.

For the tables in the experiments, only a unique hash index is created on the primary key by default. Accesses by the primary keys can be performed in constant time on average while other accesses have to scan the entire table. Also, to eliminate the influence of lock conflicts, only one worker thread is used to execute the programs in this set of experiments.

7.1.1 Effect on TPC-C

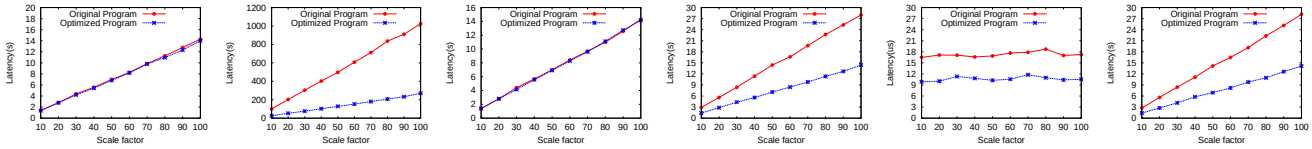
TPC-C models an order processing system and consists of five short-running transactions. We implement these transactions according to the sample programs in [29] and use them in our experiments. We compare the performance of the original and the optimized execution plans for each transaction by measuring the transaction's average latency at different scale factors. Note that, the execution time of an optimized program includes the code translation cost which is negligible compared with overall latency. The total size of tables on disk is about 70MB per scale factor.

The latency of each transaction while varying the number of warehouses is shown in Figure 9. The results illustrate that three of the five transactions, namely New Order, Payment and Delivery, benefit from the optimization of UniAD significantly. Not surprisingly, the performances of transactions Stock Level and Order Sta-



(a) New Order (b) Payment (c) Stock Level (d) Order Status (e) Delivery

Figure 9: Performance of transactions in TPC-C



(a) Delete Reservation (b) Find Flight (c) Find Open Seat (d) New Reservation (e) Update Customer (f) Update Reservation

Figure 10: Performance of transactions in SEATS

tus are comparable, because these two transactions are quite simple without much optimization space, e.g., Order Status transactions include table scans nested in the loop but they access the table $orderline$ by primary keys.

The performances of New Order and Payment transactions are improved by the reduction of tuple accesses. Both of them include updates to a small number of keys in the system. To update a tuple, the transaction first extracts the tuple's value from the table, computes its new value and updates the tuple with the new value. As the select and update statements are executed separately, two index accesses are needed to perform an update. By applying Rule 3, UniAD can merge the operations performed on the same tuple and the number of I/O operations is reduced.

The performance of Delivery transactions is significantly improved by avoiding the table scan nested in a loop. The Delivery transactions use a loop to process a batch of orders and some statements inside the loop access the table using partial keys. As only a hash index is built for each table in the experiment and the hash index does not support searches by the partial keys, sequential table scans are needed to execute these statements. By applying Rule 5, UniAD can share common table scans and Figure 9(e) shows that the optimized program yields remarkable improvement.

The results are quite interesting as TPC-C transactions are written by database experts. However, Delivery transactions exhibit poor performance when no indexes are available. It implies that even for those experts in data processing, without knowledge of the underlying storage, may also write programs which exhibit poor performance in practice. It is impractical for the programmers to write different programs for different data sets, and hence an automatic program optimization method is highly preferred.

7.1.2 *Effect on SEATS*

SEATS is an open source workload from the well-known H-Store project. It models an airline ticketing system and has been widely applied recently [9]. There are six transactions in the workload, namely Delete Reservation, Find Flight, Find Open Seat, New Reservation, Update Customer and Update Reservation. A data set generated by the standard utility is used in the experiments and the size of tables on disk is about 130MB per scale factor.

Figure 10 shows the effect on the performance of unoptimized and optimized programs by varying the scale factor. The result shows that UniAD is able to find optimization opportunities for five of the six transactions and significantly improve the performance of four transactions.

To update a customer's information, the Update Customer transaction has to probe the table customer first. The unoptimized transaction has to read two tuples and update one tuple. By merging the two table scans on the table $customer$, the optimized program is able to perform the update without another table scan. As a result, only two tuple fetches are needed and the latency of Update Customer transactions is reduced to about 2/3 of the original latency.

For New Reservation and Update Reservation transactions, UniAD achieves a 2x speedup over the unoptimized programs. A New Reservation transaction reserves an empty seat for a customer while an Update Reservation transaction updates a customer's seat to a new one. To ensure correctness, both transactions have to check whether (1) the required seat is available and (2) the customer does not have another seat on the flight. These conditions are checked in sequence — the second condition is checked only if the first condition is satisfied. The unoptimized programs hence have to scan the table $reservation$ twice. UniAD observes that the second condition can be checked together with the first one. By applying Rule 4 and Rule 3, UniAD can check the conditions simultaneously and thus eliminate the redundant table scan.

The Find Flight transaction finds available flights between two cities. The transaction first gets the nearby airports for the departure and arrival cities and then iterates over the nearby airports to find available flights. Given a nearby airport, a sequential table scan on the table $flight$ is needed to find available flights. Similar to Delivery transactions, the performance of Find Flight transactions is degraded by the nested table scan. UniAD can eliminate the nested table scan and hence improve the performance.

UniAD can also find optimization opportunities in Delete Reservation transactions. Table scans on the table $customer$ can be merged and the number of tuples a transaction accesses is reduced. But the benefit gained from the optimization is limited. It is because Delete Reservation transactions access the table $customer$ by primary keys while they have to sequentially scan the table $reservation$. In comparison with all the tuple accesses in the transactions, the tuple access we reduced is negligible.

7.1.3 *Effect on PageRank*

PageRank is a link analysis algorithm, which is used in many Web applications, and we use it as a representative of data mining programs to evaluate the effectiveness of UniAD. As PageRank cannot be easily expressed in SQL, many programmers choose to implement it in a procedural language with SQL queries. In [25], the author gave an implementation of PageRank (shown in Figure 11), which is used in our experiments.

In addition to the original program and the program optimized by the UniAD optimizer, a hand-optimized program is also used in the experiment. The hand-optimized program retrieves each node's

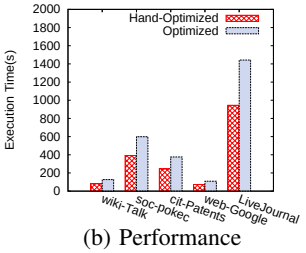
```
SELECT url INTO @url_list,
       score INTO @score_list
FROM page;
LOOP @i FROM 0 TO @url_list.size()
{
    SELECT from_url INTO @from_url_list
    FROM link
    WHERE to_url = @url_list[@i];

    @score = 0.15*@score_list[@i];
    LOOP @j FROM 0 to @from_url_list.size()
    {
        SELECT score INTO @from_score
        FROM page
        WHERE url = @from_url_list[@j];

        SELECT COUNT(to_url) INTO @from_cnt
        FROM link
        WHERE from_url = @from_url_list[@j];

        @score += @from_score / @from_cnt * 0.85;
    }
    UPDATE page
    SET score = @score
    WHERE url = @url_list[@i];
}
```

Figure 11: The PageRank program used in the experiment

score and outdegree at the beginning of the program, and hence the program can get each node's score and outdegree without scanning tables, which is the most optimal implementation.

Data Set	Nodes	Edges
wiki-Talk	2,394,385	5,021,410
soc-pokec	1,632,803	30,622,564
cit-Patents	3,774,768	16,518,978
web-Google	875,713	5,105,039
LiveJournal	4,847,570	68,993,773

(a) Data sets



(b) Performance

Figure 12: PageRank Performance

We evaluate the performance of UniAD on the pagerank programs with various datasets, the details of which are illustrated in Figure 12(a). These data sets are downloaded from the SNAP project [1]. The results are shown in Figure 12(b). Since the execution of the original program cannot be completed in six hours on the data sets which is much slower than the other two competitors, the results of the original program are omitted in Figure 12(b) for clarity.

The main cause of the original program's poor performance is the table scans nested in the loops. For each node in the graph, the program first retrieves the nodes pointing to it using the select statement. Then for each retrieved node, the program gets its score and outdegree using the statements. The procedure is repeated for each node. As a node may point to more than one node, its score and outdegree will be retrieved repeatedly.

Rule 6, Rule 2 and Rule 3 are used to optimize the original program in UniAD. By applying Rule 6, the statements in the nested loops are moved out of the loops. After that, shared table scans and corresponding loops are combined according to Rule 2 and Rule 3. To apply Rule 6, hashing maps along with the insert and search operations are introduced in the programs. Each node's score and outdegree are stored in the maps and values in the maps are used in the following statements. It resembles the method used in the hand-optimized program to improve the performance, but it is automatically constructed by the optimizer in UniAD. Because the insert and search operations performed in maps are also introduced in UniAD, which is the cost overhead, the hand-optimized program outperforms the optimized program.

Nevertheless, although the user program is not properly written, it can be efficiently executed in UniAD, yielding a comparative performance to the hand-optimized program.

[1] http://snap.stanford.edu

```
SELECT COUNT(DISTINCT(doc_id)) INTO @doc_cnt
FROM corpus;

SELECT DISTINCT(word_id) INTO @word_list
FROM corpus;

SELECT doc_id, SUM(frequency) INTO @doc_term_freqs
FROM corpus;

LOOP @i FROM 0 TO @word_list.size() {
  @word_id = @word_list[@i];

  SELECT COUNT(doc_id) INTO @doc_freq
  FROM corpus WHERE word_id = @word_id;

  @idf = @doc_freq / @doc_cnt;

  SELECT doc_id INTO @doc_inc_word_list
  FROM corpus WHERE word_id = @word_id;

  LOOP @j FROM 0 TO @doc_inc_word_list.size() {
    @doc_id = @doc_inc_word_list[@j];

    SELECT SUM(frequency) INTO @term_freq
    FROM corpus
    WHERE doc_id = @doc_id and word_id = @word_id;

    @tf = @term_freq / @doc_term_freqs[@doc_id];
    @tfidf = @tf * @idf;
  }
}
```

Figure 13: The TF-IDF program used in the experiment

7.1.4 *Effects on TF-IDF*

Term Frequency-Inverse Document Frequency (TF-IDF) is a classic method to measure a term's relevance to a document, which is popularly used in many information retrieval systems.

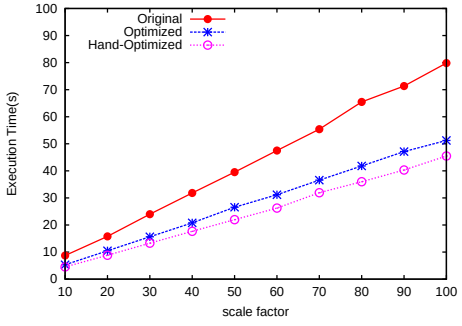


Figure 14: TF-IDF performance

In the experiment, we implement the program shown in Figure 13 to calculate the TF-IDF value of each term-document pair in the corpus. The input corpus used is randomly generated and stored in the table whose schema is $(word\_id, doc\_id, frequency)$. The number of words is fixed to 1,000 and the number of documents is 10,000 per scale factor. We also build a B^+-Tree index on the attribute $word\_id$. We again measure the performances of the original program, the program optimized by UniAD and the hand-optimized program, and the results are presented in Figure 14.

Due to the existence of the index on $word\_id$, the program in Figure 13 is a good implementation of the TF-IDF algorithm. But we still observe that the optimized program can achieve up to 40% performance promotion against the original one. By examining the

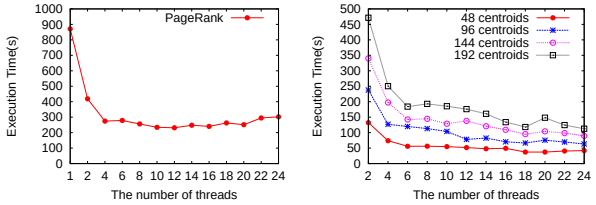
execution plan generated by the UniAD optimizer, we find that the select statements in line 13 and line 18 are merged, which eliminates the redundant tuple accesses.

The hand-optimized program exhibits a better performance as it scans the $corpus$ table to calculate each term's document frequency. As all words are listed in the $@word\_list$, index accesses are thus eliminated. Our optimizer is not able to perform such optimizations as it does not know the number of words in the $@word\_list$.

7.2 Evaluation on Parallelism

By analyzing the user programs, UniAD is able to extract logic computation that can be combined with data manipulation. The extracted logic computation might be executed in different worker threads simultaneously and the performance is improved by parallel processing. In this set of experiments, we evaluate the benefits gained from parallel program execution in UniAD. We vary the number of worker threads in the executor and measure the performance of PageRank and K-Means respectively.

We first run the PageRank program on the LiveJournal graph. To exploit data parallelism, the graph is partitioned into different partitions by hash partitioning. The number of partitions is equal to the number of worker threads and each partition is assigned to a worker thread. We measure the performance of PageRank with different numbers of worker threads and the results are plotted in Figure 15(a).



(a) PageRank (b) K-Means

Figure 15: Parallel performance

At the beginning, with the increased number of worker threads, the execution time for PageRank improves sharply. As the number of threads continues to increase, the performance remains steady and the execution time is reduced from 871s to 231s, achieving a speedup of 278%.

We evaluate K-Mean's performance with various numbers of centroids. In each step of K-Means, each node has to calculate its distance with all the centroids. The computation can be combined with the node and be executed in parallel. Thus, the amount of computation that is executed in parallel is proportional to the number of centroids. The performances of these configurations while varying the numbers of centroids are illustrated in Figure 15(b). We notice that the performance with more centroids benefits more from parallel execution. The execution time with 48 centroids can be reduced to approximate 1/3 of its original execution time while the execution time with 192 centroids is reduced to approximate 1/4.

The results exhibit that the benefits gained by parallel execution is limited by the synchronization overhead. For example, when executing a HOQ in PageRank, each node's information is gathered and written to each thread worker's local heap. When the HOQ's execution is completed, the data stored in the local heap will be flushed into the shared heap. The procedure is similar to the shuffling phase in MapReduce. As the number of worker threads increases, the amount of computation performed in each worker thread is reduced as well. However, the synchronization overhead remains the same and it finally overwhelms the benefits brought by the parallelism.

8. RELATED WORK

UniAD is closely related to many research areas including parallel data processing, programming languages, and compiler optimization. We briefly review the related work in this section.

Data processing systems. How to improve programming productivity is one of the most critical issues in computer science. Many data processing systems have been constructed to enable programmers to efficiently process data with little effort.

Relational databases have been massively successful over the past decades. Their success lies in the usage of SQL and query optimization. Users can write their tasks using SQL queries and the database can find an efficient method to execute the task. But one problem with SQL is that it is too limited to support various applications [34]. Many users have to construct very complex queries for their tasks. Object-oriented databases attempt to overcome the limitations of the relational databases by providing flexible data models and powerful development facilities, but they usually lack a suitable framework for query optimization [12, 6].

MapReduce [7] has emerged as a popular programming model for parallel data processing recently. It simplifies the development of distributed parallel applications. However, it is also criticized for its reduced functionality, considerable amount of programming efforts and lack of automatic optimization. To make MapReduce easier to use, a number of high-level languages have been developed, including Hive [28] and Pig [22].

These data processing systems all separate the execution of procedural and declarative parts in programs, which makes them fall somewhere between expressiveness brought by procedural languages and efficiency brought by declarative languages. By performing automatic optimizations across different abstractions and unifying the execution, UniAD benefits from both features of procedural and declarative programming.

Optimizations by program analysis. Some researchers proposed to apply program analysis to improve performance of data processing programs. Extracting declarative queries from imperative code has been thought to be an efficient method to improve the performance as transformed programs might benefit from the query optimization based on the relational algebra. The method is applicable for both database [19, 32, 33, 4] and MapReduce programs [14, 15]. Though these attempts try to minimize the gap between the procedural and declarative logic, the benefits they gain are limited as relational algebra is unable to match the expressiveness of procedural languages.

Other works target the applications executed in multi-tier architectures. Guravannavar et al. [13] and Cheung et al. [3] proposed different methods to reduce the overhead incurred by the communication between the applications and databases, i.e., rewrite the iteratively invoked queries into a batch form [13], and partition database programs into separate parts and execute some parts in the databases [3]. Chavan et al. [2] and Ramachandra et al. [23] studied the transformation methods to reduce the program latency by prefetching the query results.

Uniform program representation. There exist many prior studies in providing uniform intermediate representations for database programs. Some of them, such as structural recursion [1] and monoid comprehension [10], have their roots in functional programming languages. They obtain many nice properties from functional programming, e.g., they are able to abstract away detailed computation logic while retaining expressive power. But they are not suited for optimizing data processing programs as they eliminate side effects caused by I/O and shared states in programs, which makes it hard to capture the I/O efficiency of programs and limits the expression of flexible execution plans.

The other approaches include translating SQL queries into imperative code or machine code [24, 17, 21, 31, 8]. Program performance can be improved due to better data and instruction locality and the reduction of unpredictable branches. Compared with their studies, we focus on higher level optimizations and consequently we can find optimization opportunities which are ignored when programs are translated into low level code.

9. CONCLUSION

In this paper, we have presented a new system for ad-hoc data processing called UniAD. UniAD was specifically designed to simplify the programming of ad-hoc data processing tasks. Users can write their programs in procedural languages with simple queries, without the need to construct complex queries. UniAD benefits from both procedural and declarative programming by unifying the optimization and execution of user programs. We proposed a novel intermediate representation named UniQL to describe user programs. By translating user programs into HOQs, UniAD can efficiently optimize and execute user programs. We also proposed a set of optimization rules and demonstrated their efficiency through a comprehensive experimental study.

Note that, UniAD is not aimed to replace the existing data processing systems, such as Hadoop and Pregel. UniAD targets ad-hoc data processing, where the users have a relatively small set of data at hand and want to conduct a set of tests on the data immediately. UniAD frees the users from formulating their problems and thinking about the efficiency of their programs, by providing *efficient* execution for their programs even when they are *poorly* written. Once users decide to deploy a program as a standard tool, they can carefully optimize the program and run the program in a more efficient data processing system such as Hadoop.

There are several directions for future development of UniAD to provide better performance and user experience. Currently, UniAD deploys a rule-based optimizer and we would like to design a cost-based optimizer to provide more efficient optimizations. Other ongoing research involves extending the functionality of UniAD to facilitate a distributed environment.

ACKNOWLEDGEMENTS

The research is supported by the National Natural Science Foundation of China under Grant No. 61272155 and 973 program under No. 2014CB340405. Beng Chin's research is partially funded by the Singapore National Research Foundation and the publication is supported under the Campus for Research Excellence And Technological Enterprise (CREATE) programme.

10. REFERENCES

[1] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *DBPL-3*, pages 9–19, 1992.

[2] M. Chavan, R. Guravannavar, K. Ramachandra, and S. Sudarshan. Program transformations for asynchronous query submission. In *ICDE '11*, pages 375–386, 2011.

[3] A. Cheung, S. Madden, O. Arden, and A. C. Myers. Automatic partitioning of database applications. *Proc. VLDB Endow.*, 5(11):1471–1482, July 2012.

[4] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. In *PLDI '13*, pages 3–14, 2013.

[5] J. Cieslewicz, K. A. Ross, K. Satsumi, and Y. Ye. Automatic contention detection and amelioration for data-intensive operations. In *SIGMOD '10*, pages 483–494, 2010.

[6] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In *SIGMOD '92*, pages 383–392, 1992.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[8] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's memory-optimized OLTP engine. In *SIGMOD '13*, pages 1243–1254, 2013.

[9] D. E. Difallah, A. Pavlo, and P. Curino, C.and Cudré-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proc. VLDB Endow.*, 7(4):277–288, 2013.

[10] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *ACM Trans. Database Syst.*, 25(4):457–516, Dec. 2000.

[11] J. Gao, R. Jin, J. Zhou, J. X. Yu, X. Jiang, and T. Wang. Relational approach for shortest path discovery over large graphs. *Proc. VLDB Endow.*, 5(4):358–369, 2011.

[12] G. Graefe and D. Maier. Query optimization in object-oriented database systems: A prospectus. In *LNCS on Advances in Object-oriented Database Systems*, pages 358–363, 1988.

[13] R. Guravannavar and S. Sudarshan. Rewriting procedures for batched bindings. *Proc. VLDB Endow.*, 1(1):1107–1123, Aug. 2008.

[14] M.-Y. Iu and W. Zwaenepoel. HadoopToSQL: A MapReduce query optimizer. In *EuroSys '10*, pages 251–264, 2010.

[15] E. Jahani, M. J. Cafarella, and C. Ré. Automatic optimization for mapreduce programs. *Proc. VLDB Endow.*, 4(6):385–396, Mar. 2011.

[16] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., 2002.

[17] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE '10*, pages 613–624, 2010.

[18] R. Lämmel. Google's mapreduce programming model: Revisited. *Sci. Comput. Program.*, 68(3):208–237, 2007.

[19] D. F. Lieuwen and D. J. DeWitt. Optimizing loops in database programming languages. In *DBPL3*, pages 287–305, 1992.

[20] D. Maier. Representing database programs as objects. In *DBPL*, pages 377–386, 1987.

[21] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011.

[22] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *SIGMOD '08*, pages 1099–1110, 2008.

[23] K. Ramachandra and S. Sudarshan. Holistic optimization by prefetching query results. In *SIGMOD '12*, pages 133–144, 2012.

[24] J. Rao, H. Pirahesh, C. Mohan, and G. Lohman. Compiled query execution engine using JVM. In *ICDE '06*, pages 23–23, 2006.

[25] T. Segaran. *Programming Collective Intelligence*. O'Reilly, first edition, 2007.

[26] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, Mar. 1988.

[27] Stonebraker, M. and Pavlo, A. SEATS benchmark. `http://hstore.cs.brown.edu/projects/seats/`.

[28] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, Aug. 2009.

[29] TPC. TPC-C benchmark. `http://www.tpc.org/tpcc/`.

[30] P. W. Trinder. Comprehensions, a query notation for dbpls. In *DBPL-3*, pages 55–68, 1992.

[31] S. D. Viglas. Just-in-time compilation for SQL query processing. *Proc. VLDB Endow.*, 6(11):1190–1191, Aug. 2013.

[32] B. Wiedermann and W. R. Cook. Extracting queries by static analysis of transparent persistence. In *POPL '07*, pages 199–210, 2007.

[33] B. Wiedermann, A. Ibrahim, and W. R. Cook. Interprocedural query extraction for transparent persistence. In *OOPSLA '08*, pages 19–36, 2008.

[34] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI'08*, pages 1–14, 2008.

[35] B. Zou, X. Ma, B. Kemme, G. Newton, and D. Precup. Data mining using relational database management systems. In *PAKDD'06*, pages 657–667, 2006.