Practical Declarative Smart Contracts Optimization

Lan Lu*
University of Pennsylvania
lanlu@seas.upenn.edu

Tao Luo
University of Pennsylvania
taoluo71@seas.upenn.edu

Jingyi Li
University of Pennsylvania
jingyili@seas.upenn.edu

Hongxun Ding
Southern University of Science and Technology
hongxunding02@gmail.com

Brendan Massey
University of Pennsylvania
masseybr@seas.upenn.edu

Haoxian Chen*
ShanghaiTech University
hxchen@shanghaitech.edu.cn

Boon Thau Loo
University of Pennsylvania
boonloo@seas.upenn.edu

ABSTRACT

Despite all recent advancements in blockchain technology, writing efficient smart contracts still heavily relies on the expertise and ingenuity of programmers struggling with low-level languages that are hard to optimize by non-expert users. This paper presents DeSCO, a declarative framework for writing gas-efficient correct smart contracts. In DeSCO, smart contracts are written in a Datalog-style language, and a series of query optimization techniques are applied automatically to produce gas-efficient smart contracts. We identify storage operations as the primary source of overhead for smart contracts, apply traditional query optimizations to storage operations, and develop a deterministic and efficient selective view materialization approach tailored for smart contract development. In addition, our work is agnostic to the underlying blockchain platforms and platform variants by using simulation. Our evaluation, conducted on real-world smart contracts, demonstrates the efficiency of DeSCO in bridging the performance gap between automatically generated codes and prior carefully hand-tuned implementations while offering the promise of verifiability and rapid prototyping for non-expert programmers.

VLDB Workshop Reference Format:
Lan Lu, Tao Luo, Jingyi Li, Hongxun Ding, Brendan Massey, Haoxian Chen, and Boon Thau Loo. Practical Declarative Smart Contracts Optimization. VLDB 2024 Workshop: FAB.

1 INTRODUCTION

Smart contracts are digital agreements leveraging blockchain technology's decentralized and immutable nature, ensuring secure and transparent execution. Utilizing distributed consensus and cryptographic verification, smart contracts enhance trust, auditability, and tamper resistance, significantly impacting industries like finance [35], logistics [13], and healthcare [27].

*Corresponding authors

Proceedings of the VLDB Endowment. ISSN 2150-8097.

As decentralized platforms, public blockchains incentivize participants to process transactions through fees, commonly called gas costs. These fees, paid by transaction senders, are determined by the number and types of instructions executed [40]. Given the high transaction volume[1], even minor computational overheads can lead to substantial financial losses.

Today, writing efficient smart contracts still heavily relies on the expertise of programmers. Although several optimization tools exist [9, 10, 15, 19], their application is limited, particularly because these contracts require a deep understanding of low-level bytecode, making optimization challenging and error-prone.

Bugs in smart contracts have resulted in financial losses amounting to millions of US dollars [41]. Various verification tools use formal methods [11, 14, 38], static analysis [21, 22, 39], and testing [25, 36] to address these risks. However, despite their effectiveness in identifying different vulnerabilities, the complexity of the Turing-complete Solidity language [6], commonly used for smart contracts, poses smart contract verification challenges.

To enable automatic optimization and unlock further potential for more effective verification and analysis, we propose a shift towards declarative programming for smart contracts while we focus on resource optimization through query optimizations. Declarative languages offer several benefits including separating logic from implementation, enabling programmers to focus on writing high-level logic without considering the underlying implementation. This reduces errors and facilitates maintenance and verification [16]. They also enable automatic code generation and optimization [26, 28, 34], allowing even novice programmers to create efficient smart contracts.

As the first step, this paper focuses on the problem of efficient code generation for declarative smart contracts. We introduces a domain-specific Datalog language which is based on relational logic for programming smart contracts. While demonstrated to be feasible [17], Datalog implementations currently suffer from performance overheads compared to hand-written code. Our performance analysis (Section 3) identifies excessive temporary data storage as a major issue.

[1]According to Etherscan, at the time of writing this paper, Ethereum sees more than one million transactions per day, with an average transaction fee of 6.4 USD.

We propose selective view materialization to optimize storage and minimize runtime query costs. Although this technique has been extensively studied for decades, two unique challenges arise in optimizing smart contracts. Firstly, the immutable nature of smart contracts renders dynamic optimization techniques unfeasible, limiting us to static options. Secondly, previous studies on static view selections [33, 37] assume the availability of an accurate cost model. However, obtaining a static cost model for smart contracts is challenging due to the intricate mechanics of Ethereum gas calculation [40]. Despite the extensive effort to optimize smart contracts [9, 22, 30], to our knowledge, there is no accurate cost model for smart contracts.

These challenges give rise to a more precise technical problem that we tackle in this paper: *how can we statically choose the best set of materialization views without relying on an accurate cost model?*

To address the above problem, we profile smart contracts using specific execution traces within a local Ethereum emulation framework and then precisely gauge the costs associated with various candidate view materialization plans. Our approach offers two distinct advantages over conventional methods reliant on pre-defined cost models. First, the Ethereum Virtual Machine (EVM) instructions dictate the cost of executing smart contracts, which remain unaffected by environmental variables like available memory or cache performance. Second, blockchain platforms constantly evolve and develop, making it challenging to maintain a consistently accurate cost model that aligns with updates. In contrast, our profiling-based approach can be seamlessly integrated into emulation tools provided by different platforms with minimal adjustments.

Despite higher initial computational costs and time than a static cost model, the long-term financial impact justifies our profiling approach since smart contracts are long-running once deployed. We optimize profiling by identifying a subset of materialization plans guaranteed to include the optimal one, using simplification rules to eliminate unnecessary cases as in Section 5.3.

Specifically, this paper contributes to the development of gas-efficient smart contracts using declarative languages by:

Performance analysis. We perform extensive performance analysis, identifying storage operations as the key bottlenecks (Section 3).

Novel view selection. We develop novel view selection techniques for optimal materialization without relying on an accurate cost model, making it platform-agnostic and adaptable (Section 5).

Evaluation. We validating our approach through DeSCO, a Declarative Smart Contract Optimizer, that integrates novel optimization techniques and conventional database optimization strategies, demonstrating promising results in bridging performance gaps (Section 6).

2 DECLARATIVE SMART CONTRACTS

This section shows how Datalog [7] can be used for writing declarative smart contracts. Here, transaction records are treated as relational tables, and the state of contracts is defined using relational queries on these tables [17]. We use a token management smart contract example throughout this section. The smart contract supports three types of transactions: mint (to create new tokens), burn (to destroy tokens), and transfer. Figure 1 shows the transfer records and the address balance sheet as two relational tables. To illustrate how the balance sheet is calculated through a query on the transfer records, refer to the example Datalog contract presented in Listing 1, which includes four main components, as follows.

Listing 1: A Datalog contract for token management.

```
1  /* Relation declaration */
2  .decl *owner(p: address)
3  .decl *totalSupply(n: int)
4  .decl balanceOf(p: address, n: int)[0]
5  ...
6  /* Public interfaces declaration */
7  .public event recv_mint,recv_burn,
8                recv_transfer
9  .public balanceOf(1),totalSupply(0)
10 /* Function declaration */
11 .function canSend
12 /* Transaction rules */
13 mint(p,n) :- recv_mint(p,n), msgSender(s),
14     owner(s), n>0, p!=0.
15 burn(p,n) :- recv_burn(p,n), msgSender(s),
16     owner(s), p!=0, balanceOf(p,m), n<=m.
17 transfer(s,r,n) :- recv_transfer(s,r,n),
18     balanceOf(s,m),m>=n,n>0,canSend(s,r).
19 /* Inference rules */
20 canSend(p,q):-permission(p),permission(q).
21 totalOut(p,s):-transfer(p,_,_),
22                s=sum n: transfer(p,_,n).
23 totalIn(p,s):-transfer(_,p,_),
24               s=sum n: transfer(_,p,n).
25 balanceOf(p,s):-totalOut(p,o),totalIn(p,i),
26                 s:=i-o.
27 totalMint(s) :- s=sum v: mint(_, v).
28 totalBurnt(s) :- s=sum v: burn(_,v).
29 totalSupply(s):-totalMint(m),totalBurnt(b),
30                 s:=m-b.
31 ...
```

(1) Relation Declarations: This includes specifying transaction records ("transfer"), contract states ("balanceOf"), and the reception of a transaction event (" recv_transfer "). The syntax for these declarations is structured as "name(field1:type1, ...) ". When a relation has a star symbol (*) before its name, it indicates a singleton relation, meaning it contains a singular row (for instance, "owner"). There is also the option to denote primary keys, outlined within square brackets immediately following the declaration. For example, "balanceOf" designates its first column as the primary key.

(2) Relation annotations. Certain relations are annotated as public (e.g., Lines 7-9). This annotation signifies that they are accessible by regular Ethereum addresses. Line 7 also annotates three relations as event. These relations serve as the triggers for the execution of the associated event-condition-action rules, which will be explained shortly. Additionally, there are relationships that can be designated as functions (for example, Line 11), which refers to relationships identified for on-demand computation rather than storage in memory (further elaborated in Section 5.3).

(3) Transaction rules. The syntax and semantics of rules are based on Datalog, except that recursions are prohibited for efficiency [19]. It makes a distinction between two kinds of rules: transaction rules and inference rules. Transaction rules can be interpreted as event-condition-action rules, where only particular events can trigger the evaluation of the rule. Rules at Lines 13-18 are examples of transaction rules. Each transaction rule contains one and only one annotated event relation in the body, which is exclusively triggered upon receiving a corresponding transaction request. The rest of the rule body determines whether the transaction can be committed.

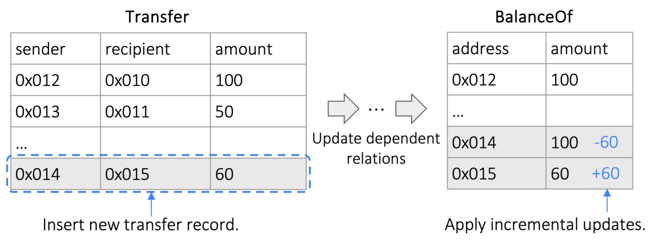


Figure 1: Relational views of transaction records and contract states, and their updates on committing new transactions.

Instruction type		Gas
Compute		3-10
Memory		3-12
Storage	Load	2,100/100
	Store (init)	22100/20,000
	Store (update)	5,000/2,900/100
Other		vary by data sizes

Table 1: EVM bytecode gas cost model.

Take the rule at Line 13 for instance, the "recv_mint(p,n)" predicate served as the event trigger, because relation "recv_mint" is annotated as a public event at Line 7.

(4) Inference rules. Inference rules are interpreted as invariants: updates to any of the body relations will trigger rule evaluation. For instance, Lines 20-30 define different relations as declarative queries over other relations.

Given the above specification, executable codes can be generated to maintain these view relations in an incremental manner: each committed transaction involves the insertion of a new row into the corresponding transaction record table, and incremental updates are applied to all the dependent views (contract states) recursively.

Figure 1 illustrates an example of committing a " transfer " transaction. It is interpreted as a new row inserted into the corresponding table. It then updates the dependent relations: " totalIn " and "totalOut", which trigger updates to "balanceOf". Through this process, declarative smart contracts automatically maintain the relational invariants, which are defined as Datalog rules, as new transactions are introduced.

3 CASE FOR OPTIMIZATIONS

In several permissionless blockchains, e.g., Ethereum, resource quotas (in the form of *gas* consumptions) are imposed to limit resource utilization. The goal is to minimize gas consumption for each smart contract transaction execution.

We motivate the need and potential for query optimizations through a case study on gas consumption profiling of a representative Ethereum smart contract. Our example is based on the transfer transaction of the BNB Token smart contract [4]. BNB Token is based on a popular cryptocurrency and this particular transaction bears similarity to the wallet example presented in Section 2.

It should be noted that the EVM byte-code gas cost model, as shown in Table 1, reveals a significant disparity in costs between reading and writing to storage compared to compute instructions. Notably, initiating storage values incurs extremely high gas expenses when a storage slot is set from zero to a non-zero value.

Instruction type		Datalog	Ref.
memory	read	36	6
	write	195	51
	others	1,518	1,756
storage	read	31,900	4,500
	write	13,900	5,800
compute		6,769	1,986
transaction fee		21,570.8	21,570.8
all		75,888.8	35,669.8

Table 2: Gas breakdown of the BNB Transfer transaction.

Table 2 presents a detailed breakdown of gas costs per transaction, categorized by instruction types for the Token BNB contract. This analysis shows that storage operations are the dominant factor in overall gas usage, while the transaction fee is a flat fee for all transactions. In addition, Baseline without optimization shows significant inefficiencies in gas utilization compared to the reference implementation (Ref.), which is meticulously hand-optimized.

Further investigation into the Solidity code produced by the Baseline compiler uncovers another issue: *an excessive number of function invocations.* As the number of relations grows, it results in complex chains of function calls and updates to those relations. Consequently, a large amount of memory operations are needed to pass arguments between functions and synchronize the updated values with the corresponding relations, leading to extra gas consumption.

Similar patterns of inefficiency are observed in other smart contracts as well, emphasizing the need to address these challenges. Our performance analysis yields two key insights for optimization.

(1) Reduce storage manipulation and usage. Traditional incremental Datalog follows conventional wisdom by materializing all intermediate tables [17]. These tables are then incrementally recomputed each time new facts (transactions) arrive. However in smart contracts, the gas consumption gap between computation and storage operation is significant, in part because storage in smart contracts is expensive (particularly if the blockchain ledger is involved, since that requires running a bandwidth-expensive consensus protocol).

Consequently, we can consider a view materialization strategy which selects only a subset of intermediate tables for materialization, to save gas by minimizing costly storage-related instructions during the view maintenance process and replacing them with less expensive computational operations during query execution. To classify materialization strategies free of such redundancies, we introduce the notion of "minimal form" in Section 5.2. Note that multiple methods exist for removing these redundant relationships, leading to different "minimal" choices of the original redundant plan. Thus, the selection of the optimal view entails a search process. In Section 5.4, we will introduce further techniques for narrowing down the search space. This is intended to accelerate both the processes of retrieving plans and executing simulation tests.

(2) Arithmetic simplification enables view elimination. Through arithmetic simplification, certain intermediate views can be eliminated by simplifying and removing variables in arithmetic expressions. This leads to fewer required reads of the views and consequently decreases the number of instructions needed for reading and updating the storage associated with these views.

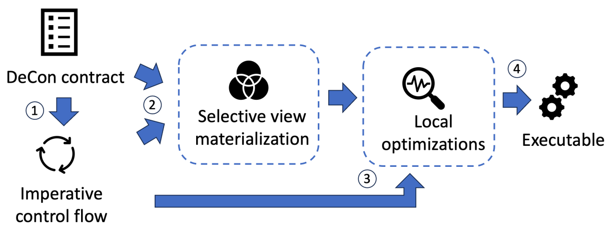
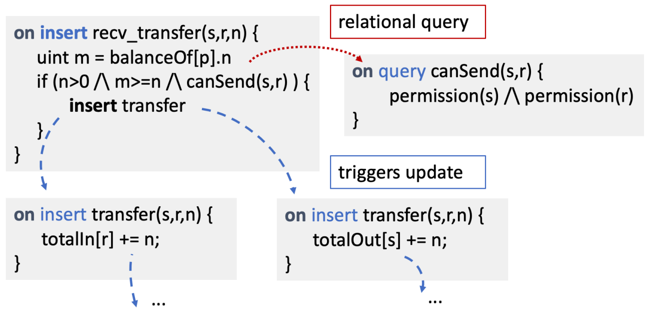


Figure 2: Overview of DeSCO's optimization flow.



```
on insert recv_transfer(s,r,n) {
    uint m = balanceOf[p].n
    if (n>0 ⋀ m>=n ⋀ canSend(s,r) ) {
        insert transfer
    }
}

on query canSend(s,r) {
    permission(s) ⋀ permission(r)
}

on insert transfer(s,r,n) {
    totalIn[r] += n;
}
...

on insert transfer(s,r,n) {
    totalOut[s] += n;
}
...
```

Figure 3: Imperative control flow.

4 OVERVIEW

An overview of the optimization flow in DeSCO is provided in Figure 2. This flow comprises two steps critical to optimization; first, selective view materialization and second, local optimizations. The overall code generation and optimization flow are as follows: **(1)** DeSCO converts declarative rules in a Datalog contract into an imperative control flow, which validates incoming transactions and maintains all declared relations when a valid transaction is committed. Unlike conventional Datalog engines, DeSCO further optimizes the imperative control flow by identifying differentiable fragments in the rules to streamline the imperative control flow. **(2)** By leveraging information from the original Datalog contract and generated imperative control flow, DeSCO performs selective view materialization, to selectively store a subset of declared relations in persistent storage. It is crucial as it determines the program's overall structure and significantly contributes to gas savings. **(3)** Once the imperative control flow and the view materialization plan are established, DeSCO applies various local optimizations to each update procedure. These optimizations, including indexing and read projection, collectively enhance the efficiency of the implemented smart contract. **(4)** Finally, DeSCO compiles the execution plan to Solidity and generates executable code for the Ethereum platform.

We next illustrate these four steps using the token management contract, presented in Listing 1.

Imperative control flows. Figure 3 depicts an imperative control flow example, where each code block is generated from a rule r and a corresponding update to a relation in r's body. The top-left code block is generated from the rule at Line 17. When the contract receives a transfer request with parameters, it initially searches the "balanceOf" table for a tuple "(p,m)" that satisfies two constraints: "p==s" and "m>=n". If such a tuple is found, it then checks two conditions, which are directly derived from the

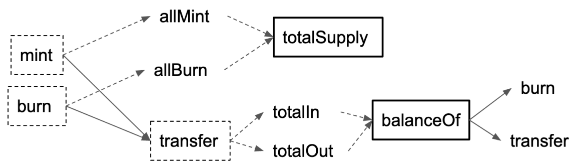


Figure 4: Relation dependencies in the Wallet contract.

remaining part of the rule body. Since the relation "canSend" is defined to be computed on demand, the predicate "canSend(s,r)" is implemented as a query on other relations, depicted by the red dotted arrow. The code for this query is displayed in the top right portion of the figure, and it is generated from the rule at Line 20 in Listing 1. If all conditions are met, it inserts a new "transfer" tuple, which triggers updates to other dependent relations, depicted by the blue dashed arrow. The updates are shown in the bottom portion. These updates are recursively triggered until no dependent relations can be updated. The remaining part of the update procedure is omitted for conciseness.

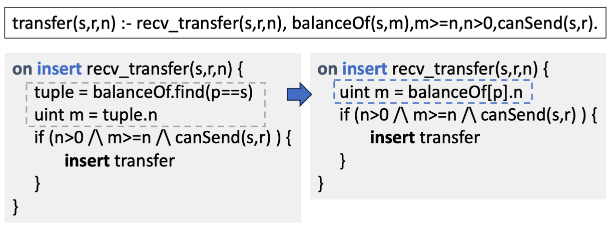
Selective view materialization. With the imperative code, DeSCO chooses a subset of relations to be materialized. Recall from Section 3 that the high storage cost of public permissionless blockchains requires a storage-efficient view materialization plan. This distinction calls for a fundamental overhaul of the materialization strategy, departing from traditional approaches where storage efficiency is not a primary focus and is often sacrificed for time efficiency.

The goal of view materialization is to answer all relational queries efficiently in the imperative control flow. Intuitively, a relational query $R(\bar{X})$ can be answered in two ways: (1) directly read from memory, for the case where relation R is materialized; or (2) derive from its defining rules, for the case where R is not materialized. Identifying all possible view materialization plans requires analysis on the dependency among different relations.

Figure 4 visualizes the dependency among relations in the wallet contract. For instance, "totalSupply" can be derived from "allMint" and "allBurn". This suggests that, in order to answer queries to "totalSupply" (declared as a public query interface at Line 9), we do not need to store all of the relations: either storing only "totalSupply", or storing "allMint" and "allBurn" is sufficient.

One may think that storing only query results can solve this problem, obviating the on-demand computations and storage for intermediary relations. However, this is not always the most efficient, considering the view maintenance cost. Due to the dynamic nature of smart contracts, characterized by frequent transaction commits, the associated view update procedure could incur substantial gas costs. To see this, consider the "canSend(p,q)" relation defined at Line 20 in Listing 1: "p" can send to "q" if both of them have permission "permission(p)", "permission(q)". Maintaining "canSend(p,q)" means that on every update to the "permission", one has to iterate through the "permission" table to generate new "canSend" tuples. In this case, the best strategy is to materialize "permission" instead, and evaluate queries to "canSend(p,q)" on demand.

These two examples highlight the primary and contradictory factors influencing view materialization decisions: *query execution cost* and *view maintenance cost*. The proactive maintenance of query



```
transfer(s,r,n) :- recv_transfer(s,r,n), balanceOf(s,m),m>=n,n>0,canSend(s,r).
```

```
on insert recv_transfer(s,r,n) {
   tuple = balanceOf.find(p==s)
   uint m = tuple.n
   if (n>0 /\ m>=n /\ canSend(s,r) ) {
        insert transfer
   }
}
```

```
on insert recv_transfer(s,r,n) {
   uint m = balanceOf[p].n
   if (n>0 /\ m>=n /\ canSend(s,r) ) {
        insert transfer
   }
}
```

Figure 5: Indexing and read projection.

results can reduce query execution costs but may incur high view maintenance expenses due to substantial storage overhead and frequent storage operations. Conversely, maintaining a minimal set of relations can reduce view maintenance costs but may increase overhead during on-demand query execution.

Section 5 introduces our algorithm to select the most gas-efficient view materialization plan. In particular, it first applies arithmetic optimization on the imperative control flow, then prunes the view materialization space following certain elimination rules, and finally estimates the gas cost for each remaining viable view materialization plans via profiling on a local simulation blockchain.

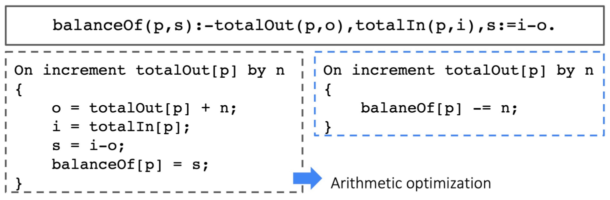
Local optimizations. Given the imperative control flow, DeSCO further optimizes gas usage within each code block. Figure 5 illustrates the optimized block derived from the left side of Figure 3. Firstly, an index is added to the "balanceOf" relation, thereby economizing on iterations to locate tuples with the address "s". Secondly, read projection is applied to the reading of the "balanceOf" tuple, where only the second column is read, circumventing the need to read the entire tuple and subsequently project the second column. These optimizations contribute to a more resource-efficient maintenance of contract states.

5 SELECTIVE VIEW MATERIALIZATION

In this section, we first introduce view elimination techniques through arithmetic expression simplification. Next, we establish the validity of a materialization plan based on the optimized imperative control flow. Then, we introduce an algorithm that systematically identifies all minimal valid materialization plans and determines the most gas-efficient option through profiling.

5.1 View Elimination

The imperative control flow programs are responsible for maintaining the relations defined by the Datalog rules, as depicted in Figure 3. The compilation process that generates the imperative control flow is divided into two phases. First, it identifies the set of potential updates that may occur while the smart contract is being executed, such as the insertion, deletion or update of a relational tuple. And second, once the set of updates is determined, it converts the rules into imperative maintenance procedures which generally consists of lookup of relational tuples in the rule body, validating the rule constraints, and generating updates to the rule head. Note that the updated rule head can further trigger the updates to following associated rules. Given this control flow, we introduce arithmetic optimizations to further eliminate intermediate views and variables.



```
balanceOf(p,s):-totalOut(p,o),totalIn(p,i),s:=i-o.
```

```
On increment totalOut[p] by n
{
    o = totalOut[p] + n;
    i = totalIn[p];
    s = i-o;
    balanceOf[p] = s;
}
```

```
On increment totalOut[p] by n
{
    balaneOf[p] -= n;
}
```

Figure 6: Incremental update without (left) and with (right) arithmetic optimization.

Consider the rule at Line 25 in Listing 1, which is displayed at the top of Figure 6 for ease of reference. This rule defines the balance of an address "p" as the difference between the sum of its incoming transfers "i" and the sum of its outgoing transfers "o". Let us assume a new transfer "transfer(p,q,10)" is committed, leading to an increment of "totalOut[p]" by 10. If we were to interpret this rule as a regular join rule, as depicted in the bottom-left portion of Figure 6, the incremental maintenance procedure would involve retrieving the sum of incoming transfers of "p", storing it in local memory as "i", and then computing the new balance by subtracting "o" from "i". However, upon analyzing the arithmetic operator, we can deduce that the update to "n" is −10. This insight gives rise to a simpler update approach, as demonstrated in the bottom-right section of Figure 6: we can directly decrement "balanceOf[p]" by 10, eliminating the need for storage retrieval and local computation.

In essence, we are trying to identify a differentiable fragment of the update procedure that allows us to calculate the update to the dependent relation solely based on the difference in the updated body relation, without referring to the concrete value of it. In the above example, "totalOut[p]" is such a differentiable fragment. Since this update is agnostic to the concrete value of "totalOut", and there are no other places that read the "totalOut" relation, we can eliminate "totalOut" from the imperative control flow. Similarly, " totalIn " can also be eliminated. Overall, the idea of differentiable fragment here is general and can be extend to addition, subtraction, aggregation, count, and even multiplication.

DeSCO identifies such differentiable fragments by checking all update procedures to all rules, and eliminates the corresponding read operations to the corresponding relation. After this pass, DeSCO eliminates relations that are not read by any update procedure from the imperative control flow.

5.2 Materialization Plan Validity

A materialization plan refers to a set of relations to be materialized (maintained in persistent storage). The plan is valid if and only if all relational queries in the imperative control flow can be executed using the materialized views.

Executable relations. The executability of a relation can be determined by the existence of the queried tuples in the associated relational table, or the fact that it can be derived from other executable relations following the associated rules. Specifically, the executability of a transaction or query is established on the notion of *executability of a relation*. A query is executable if the associated relation is executable, and a transaction is executable if all relational queries in its imperative control flow are executable, while the imperative control flow can be optimized by arithmetic simplification.

For instance, the relation " recv_transfer " is annotated as public at Line 7 of Listing 1. The rule outlined at Line 17 defines the start of the execution flow of a transfer transaction: it is triggered by a " recv_transfer (s, r, n)" event and subsequently evaluates the executability of the body predicates, namely "balanceOf(s, m)" and "canSend(s, r)". If the transaction parameter is legitimate ("n > 0"), the sender "s" can send to the receiver "r", and the sender has a sufficient balance ("m >= n"), the transaction proceeds to update dependent relations. Specifically speaking, a " transfer " tuple is inserted, which triggers the updation of " totalIn " and "totalOut" according to Lines 21 and 23. This further triggers updating "balanceOf" at Line 25, which leads to the end of this execution flow. Then based on the arithmetic optimization, " transfer ", " totalIn ", and "totalOut" can all be simplified from the imperative control flow. Note that considering the sum operator on " transfer " records, the updated amount of the body relation each time can be understood as the concrete value of the newly inserted " transfer " record while we can fulfill the aggregation using only this updated amount with the arithmetic optimization.

This example illustrates that the relations "balanceOf" and "canSend" are necessary to be executable to determine the legitimacy of the received transaction. Suppose we have a view materialization plan that materializes neither "balanceOf" nor the two relations "allMint" and "allBurn" that can establish "balanceOf"; in that case, this view materialization plan is deemed invalid since it cannot answer relational queries to "balanceOf", thereby compromising the executability of the transaction.

From these examples, we see how the executability of a relation is connected with the functionality of a declarative smart contract.

Valid materialization plans. Given the notion of executability of relations, we define the *validity of a view materialization plan*: Let P be the set of relations obtained by the imperative control flow, a materialization plan V is valid if and only if for every relation q in P, q is executable based on the materialization plan V.

Minimal Materialization Plan. Minimal materialization plan is defined as a special set of relations which (1) is a valid materialization plan, and (2) can not form a valid materialization plan if we remove any relation(s) from it.

5.3 Space Pruning Patterns

Constructing a precise cost model for smart contracts is challenging. However, with insights from Ethereum smart contract execution and cost models, we can differentiate between efficient and less efficient view materialization plans. Specifically, identifying patterns that highlight inefficiencies allows us to eliminate such plans during the enumeration, significantly narrowing the search space.

(1) Relation minimality. Non-minimal view materialization plans are converted to their minimal forms by removing redundant relations from the materialization plan.

(2) Join complexity. Relations defined by rules that involve considerably costly join operations are not materialized. DeSCO employs a heuristic to estimate the join cost: if a join operation cannot benefit from index optimization and requires looping through a table, it is considered expensive and thus unsuitable for runtime maintenance. Illustrated by the example in Section 4, materializing the "canSend(p,q)" relation incurs significant overhead due to each insertion into the permission relation triggering a loop through "permission" itself. Conversely, during runtime querying, by indexing the permission relation, the query "canSend(p,q)" can be efficiently resolved with just two read instructions. Therefore, DeSCO identifies such expensive joins and opts to represent the head relations, like "canSend(p,q)", as functions, as detailed in Section 2.

(3) Aggregations. On the contrary, relations defined by aggregation rules are generally deemed expensive to be queried on run-time and are always maintained incrementally. A detailed example is provided in Sec 5.4 to explain the usage of this rule.

(4) Transaction relations. Transaction relations (with the "recv_" prefix) are reserved as triggers for incoming transaction requests, and thus are not considered for materialization.

5.4 Minimal Materialization Plan Enumeration

In this section, we explain how our algorithm can generate materialization plans. Given the input smart contract, we enumerate every valid minimal materialization plan in the simplified serach space and return the set of such minimal plans as the output.

The algorithm starts with a minimal base materialization plan and creates a supplementary data structure known as the replacement graph G. This graph G is built based on the dependency information obtained from the set of all contract rules P. Then this algorithm enumerates all minimal alternative materialization plans in a breadth-first search fashion based on the replacement graph G. Starting from the initial choice, namely the minimal base materialization plan, it iteratively generates alternative plans for each potential materialization plan by substituting relations with others that can calculate them as needed, based on the dependency information provided by the replacement graph G. Finally, the set of alternative replacement materialization plans is also turned into their minimal forms which are then included in the final solution.

Base materialization plan. This base plan represents the read-only approach: storing all relations that could be accessed during transaction execution or public queries, without any on-demand computation. It is then reduced to the minimal form.

For instance, the base materialization plan for the wallet contract in Listing 1 initially includes relations like "totalSupply", "balanceOf", and "permission". Through arithmetic optimization, relations such as "totalMint", "totalBurn", " totalIn ", and "totalOut" are simplified from the control flow and therefore not included in the base plan. Moreover, following the discussed pruning techniques, "permission" is materialized in the base plan instead of "canSend", since the latter is more efficient to be queried on demand than incrementally maintained in storage. Additionally, transaction records like "mint", "burn", and " transfer ", which are inserted in response to incoming transactions and not revisited later, are also omitted from the base plan.

Construct replacement graph. We now showcase the process of constructing the replacement graph. Given rules P in a Baseline program, for each relation v in P, the set of $v's$ defining rules R consists of rules whose head relation is v. Generally, it is feasible to materialize all the defining relations of v and calculate relational queries to v whenever necessary. Consequently, for each defining relation u of v, an edge from u to v is included in the replacement graph G to signify that relation v can be computed using information from relation u. Figure 7 shows the replacement graph constructed from the example contract in Listing 1.

For example, consider the following rule that defines the relation "totalSupply", taken from Line 29 in Listing 1:

```
totalSupply(s):- totalMint(m), totalBurnt(b), s:=m-b.
```

It defines "totalSupply" as the difference between the sum of the minted amount ("totalMint") and the sum of the burnt amount ("totalBurn"). Based on this rule, two edges are added to the replacement graph: ("totalSupply", "totalMint") and ("totalSupply", "totalBurn"). Consequently, it is possible to choose not to materialize "totalSupply" but instead materialize "totalMint" and "totalBurn". When queried, the value of "totalSupply" can be computed from these two materialized relations. Note that even "totalMint" and "totalBurn" are not required to be executable in the imperative control flow, they can still be chosen to be materialized to support the executability of "totalSupply" by computing.

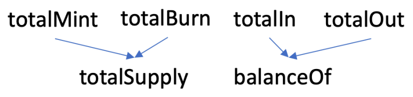


Figure 7: Replacement graph of the Wallet contract.

Given the replacement graph, we now can generate alternative plans by substituting a relation in an existing materialization plan with its defining relations. For example, given a plan V that includes "totalSupply", this algorithm can replace "totalSupply" with "totalMint" and "totalBurn" as a new materialization plan.

5.5 Selecting Optimal Plan

Given all valid minimal materialization plans, DeSCO picks the most efficient one through profiling on synthetic transaction traces. The details of synthetic trace generation are described in Section 6.

The profiling trace provides an approximation of the gas expenditure for each type of transaction. However, there is still an important question that needs to be addressed: how to optimize across all transaction types? For instance, the ERC20 standard includes various transaction interfaces, but most ERC20 contracts today are dominated by the " transfer " transaction, while other transaction types like "mint" and "burn" are executed less frequently. If the objective is to minimize the overall gas cost on the actual deployment workload, the optimal approach would be to prioritize the materialization plans that incur the least gas overhead for the "transfer" transaction.

Given the disparity of execution frequency among different transaction interfaces, DeSCO accepts the expected execution frequency of each transaction type as an optional input. If the expected transaction frequency is given, DeSCO employs the weighted average of the transaction gas cost as an estimation of a smart contract efficiency metric. Otherwise, the average is used instead, assuming that the transaction types follow a uniform frequency distribution.

6 EVALUATION

We evaluate our optimization techniques by comparing the end-to-end gas cost with several baselines. The goal is to ensure that the code generated by the automatic optimizer can match the performance of hand-optimized reference implementations. We implement the optimizer on top of the DeCon compiler. The optimizer is evaluated on the 13 most popular token contracts sourced from Etherscan [1] and OpenZeppelin [3]. All benchmarks and complete evaluation results are available online [2]. Experiments are run on Ganache, a local Ethereum simulator from Truffle [5].

6.1 Profiling Set Up

Contract benchmarks. We select 13 popular real-world contracts that can be supported by Datalog semantics. We prepare and adjust the Datalog logic to let its declarations of public interfaces match with the corresponding open-source contract, such that we can evaluate the Datalog-generated versions and the reference implementations using the same workload.
Synthetic workload. We generate synthetic workloads for each benchmark contract transaction type (e.g., "mint", "transfer", etc.), with randomly generated parameters. The main reason is that via experiments we found the actual parameter values of an incoming transaction generally have little impact on the gas consumption which is dominated by the number of storage operations. Besides, many of the contracts lack enough real-world data from the blockchain platform like Ethereum. To capture the stable gas cost, we use the first transaction as the warm-up call and average the gas of the subsequent 10 transactions as the final estimation.
Transaction Frequency. For each contract, we count the actual number of occurrences of each transaction type from the most recent 10,000 transaction records on Etherscan, and converted them into frequencies. As in Sec 5.5, by combining these frequencies with each transaction's gas cost, DeSCO can calculate a weighted average gas consumption of the contract. In the experiment, if the contract's frequency data is available and relatively stable in the period, we adopt this method of frequency-weighted averaging. Otherwise, we directly use the average of all transactions to select the best DeSCO optimized version and compare it with the other baselines.
Profiling. In the experiment, we compare three approaches:

- **Incremental Datalog:** Vanilla incremental Datalog execution without optimizations.
- **DeSCO:** Smart contracts are written in Datalog and then the full suite of optimizations are applied.
- **Reference:** We select popular Solidity programs that are active in the open source and are supported by Datalog semantics, either from Ethereum or previous works. These programs have extensive manual optimizations performed by multiple stakeholders.

Note that optimizations of read projection and indexing are not our main focus of research, so we implement these techniques in all the compiler generated versions including *Incremental Datalog*. In addition, since the triggers and records of each transaction are normally not stored by the blockchain, they are not in the domain of materialization and are also excluded from the *Incremental Datalog*.
Profiling cost. The profiling cost for DeSCO is affordable. Firstly, the materialization plan generation is one-time and would finish within 1 minute. Besides, the number of minimal materialization plans of each contract is often limited, e.g., generally less than 10 and up to 16 in BNB contract, due to the small number of relations

Contract	Datal.	Ref.	DeSCO	IR(Dl)	IR(Ref)
Erc20	67	35.6	37.2	44%	-4%
Erc777	81.44	44.78	43.89	46%	2%
BNB	72.99	35.25	35.53	51%	-1%
Controllable	63	35.22	35.78	43%	-2%
Link	69.46	34.73	35.81	48%	-3%
Matic	69.13	37.2	37.06	46%	0
Shib	46	30	31	33%	-3%
Wbtc	56.56	37.98	35.4	37%	7%
Auction	60.33	46.67	46.67	23%	0
TokenPartition	126	47.67	45.33	64%	5%
LtcSwapAsset	67.8	40.8	38	44%	7%
Wallet	62	36	36	42%	0
VestingWallet	32	28	32	0	-14%

Table 3: Gas consumption of all contracts on three approaches.

in each contract and blockchain's lack of support for complex and thus expensive contracts. So DeSCO can generate all versions of solidity programs for all contracts within several minutes. Specifically, generating one solidity program would cost approximately one second. Furthermore, for each solidity version of a contract, running simulation tests using the synthetic workloads for all its transaction types would take just a few seconds, considering that testing one type of transaction is generally about 3 or 4 seconds.

6.2 Main results

Table 3 shows that DeSCO matches the performance of the open source hand-optimized *Reference*, significantly outperforming *Incremental Datalog*. The first three columns present the gas cost (unit: thousand (k)) of all our contracts under the *Incremental Datalog*, *Reference*, and DeSCO. The last two columns compute the improvement rate of DeSCO over *Incremental Datalog* and *Reference* respectively.
Improvement over *Reference* implementation. For some contracts (e.g., Erc777 and Wbtc), DeSCO slightly outperforms the carefully hand-tuned *Reference*. This is because DeSCO reads and materializes only the necessary relations. Considering the case where a non-expert Datalog programmer may define a lot of auxiliary relations, DeSCO's selective view materialization yields great benefit by consistently materializing the minimal number of relations. In addition, DeSCO consistently applies read projections, which is neglected in some *Reference* implementations.
Cases where DeSCO is less efficient than *Reference* implementation. We identify contracts (e.g., Erc20 and BNB) where DeSCO performs a bit worse than *Reference*. In this case, although DeSCO does not materialize more relations than *Reference*, there are some code-level optimizations to be leveraged. For instance, *Reference* implements self-increment and self-decrement of variables using $a += b$ and $a -= b$, while in the compiler, we always generate $a = a + b$ and $a = a - b$ which cost extra gas based on a controlled experiment. Function inlining is another contributing factor to the better performance of *Reference*. While *Reference* implementation has the complete transaction logic written within one or two function(s), DeSCO generates update functions for each Datalog rule, and executes the transaction logic in a chain of function invocations.

Note that these features are not the primary focus of this project, as their impact on the contract's gas cost is deemed insignificant. Moreover, they are well-known standard optimization strategies that can be incorporated into future versions of DeSCO.

Instruction type		Datalog	Ref.	DeSCO
memory	read	36	6	6
	write	195	51	63
	others	1,518	1,756	1,518
storage	read	31,900	4,500	4,300
	write	13,900	5,800	5,800
compute		6,769	1,986	3,092
transaction fee		21,570.8	21,570.8	21,570.8
all		75,888.8	35,669.8	36,349.8

Table 4: Gas breakdown of the BNB Transfer with DeSCO.

For the "vestingWallet" contract, DeSCO shows little improvement over *Incremental Datalog*, and has a bigger performance gap compared to *Reference*. Upon further investigation into the compiled Datalog program, we find there are just a few relations declared, which are all required to be materialized, bringing limited optimization space for DeSCO's selective view materialization technique. On the other hand, "vestingWallet" uses a combination of conditions, thus based on the true or false value of each condition, there are multiple if-else branches. However, in different branches, the contract is reading the same relations. *Reference* reads all the relations once before the executing of all the following branches, while in Datalog semantics, each rule can only represent one branch, causing DeSCO to read all the relations again in the functions generated for different rules representing different branches, and thereby consuming more gas. In summary, this performance degradation is primarily affected by the limitations of Datalog semantics, while it is possible to improve it by exploring rule sharing later.
Gas breakdown study after optimization. As depicted in Table 4, DeSCO significantly reduces storage compared with *Incremental Datalog*, matching that of *Reference*. Computation cost is also reduced by up to 50% from the *Incremental Datalog* baseline. The computation overhead over *Reference* is caused by code-level optimization discussed earlier. While the difference of the *memory other* gas consumption between *Reference* and DeSCO is due to different event logging mechanisms, this has little impact on the result analysis. While we take BNB transfer transaction as an example, other contracts and transactions hold these trends in gas reduction. This result demonstrates the effectiveness of DeSCO.

7 RELATED WORK

Smart contract optimization. Extensive studies have been conducted on optimizing Solidity code or the underlying EVM byte code. For instance, (1) super optimization of the EVM bytecode [10], (2) anti-pattern detection [18, 22] in the source code, and provide partially automatic optimization, and (3) program data structure transformation [19], which is automated via program synthesis techniques, with extra annotation on the desired data structure. While these tools are applicable to general smart contracts, they mainly focus on lower-level local optimizations and omit more global optimizations that a higher-level abstraction can provide. Moreover, a declarative programming approach focuses on executable specifications (which have other benefits) such as verifiability.
Declarative contracts. In the legal domain, declarative languages are used to implement and reason about contracts [8, 23, 31]. The prior work largely focuses on using logic as a specification language to uncover legal conflicts [12]. DeCon [17] is a declarative programming language for implementing smart contracts. Its key

innovation is the introduction of a declarative view of smart contracts. However, unlike our work, the previous studies did not address the execution efficiency of declarative languages, nor did they consider how to leverage database query optimization techniques. Our work is the first to address the execution efficiency of these languages and to consider how to leverage database query optimization techniques.

Datalog for smart contract analysis. In addition to programming, Datalog has also been applied to smart contract analysis. Securify [39] uses Datalog to identify security patterns in smart contract bytecodes, and Ren et al. [32] analyze based on semantic facts extracted from contract source code.

View selection. Materialized views are well studied in the databases field [29]. Research in this area involves tackling various problems, such as rewriting queries using materialized views [24] and selecting appropriate views for materialization [20]. View selection algorithms can be classified based on several factors, namely query workload, objective functions and constraints, search space of views, and search algorithms for view materialization. Our view selection approach has the following key differences: (1) targeting an OLTP-like Datalog workload instead of OLAP; (2) reducing gas consumption by minimizing storage cost; and (3) representing the search space as a logical graph because each view has a unique dependency on other views/relations defined by Datalog rules.

8 CONCLUSION

This paper proposes a declarative approach to smart contract optimization, using high-level programming to generate gas-efficient implementations automatically. Our measurements showed that storage operations are the main Ethereum smart contract overhead. We achieved promising results in real-world contract evaluations by applying traditional query optimizations and introducing novel selective view materialization. This approach closes the performance gap between automated and expert-tuned code, providing efficient solutions and easier development for non-experts. Targeting at non-expert programmers, our idea can even facilitate the use of other languages like SQL for writing smart contracts, and provide insight for improving other open-source Solidity contracts without carefully manual optimization.

As immediate follow-on work, we will enhance and explore more techniques applied to our declarative smart contract optimizing system while expanding the declarative language's capability to express all types of smart contracts, including the recursive ones and non-independent contracts which are related with each other. Except for strong practicality, we also aim at a complete system with rigorous formal definitions and proofs. Despite the challenges mentioned in Sec 1, in the long term, having an accurate gas cost model will be most effective at finding optimal executiion plans. To this end, we plan to develop an accurate gas cost model based on predefined formula without actual simulation that is both efficient and compatible with the evolving Ethereum platform.

Our work lays the foundation for several exciting avenues of future work. We intend to explore the impact of declarative programming in the context of permissioned blockchains, where batch parallelism could greatly enhance smart contract performance. Multi-query optimizations where similar transactions share overlapping work can potentially further reduce resource utilization and, in fact, enable our approach to outperform hand-written implementations – a task made feasible through our use of Datalog. These future directions hold great promise for advancing the field of declarative programming for smart contract development and enhancing the overall efficiency and effectiveness of decentralized applications.

REFERENCES

[1] [n.d.]. Etherscan. https://etherscan.io/tokens. 2023.
[2] [n.d.]. Evaluation benchmarks and results. https://github.com/sigmod25/sigmod25/blob/main/evaluation_results.xlsx.
[3] [n.d.]. OpenZeppelin. https://github.com/OpenZeppelin/openzeppelin-contracts. 2023.
[4] [n.d.]. Token BNB. https://etherscan.io/token/0xB8c77482e45F1F44dE1745F52C74426C631bDD52.
[5] [n.d.]. Truffle Suite. https://trufflesuite.com. 2023.
[6] 2023. Solidity Programing Language. https://soliditylang.org/.
[7] S. Abiteboul, R. Hull, and V. Vianu. 1995. Datalog. In *Foundations of databases*.
[8] S. Agarwal, K. Xu, and J. Moghtader. 2016. Toward machine-understandable contracts. *Artificial Intelligence for Justice* (2016).
[9] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio. 2020. Gasol: Gas analysis and optimization for ethereum smart contracts. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.
[10] E. Albert, P. Gordillo, A. Rubio, and M.A. Schett. 2020. Synthesis of super-optimized smart contracts using Max-SMT. In *International Conference on Computer Aided Verification (CAV)*.
[11] Xiaomin Bai, Zijing Cheng, Zhangbo Duan, and Kai Hu. 2018. Formal Modeling and Verification of Smart Contracts. In *Proceedings of the 2018 7th International Conference on Software and Computer Applications (ICSCA '18)*. Association for Computing Machinery, New York, NY, USA, 322–326. https://doi.org/10.1145/3185089.3185138
[12] S. Batsakis, G. Baryannis, G. Governatori, I. Tachmazidis, and G. Antoniou. 2018. Legal Representation and Reasoning in Practice: A Critical Comparison. In *International Conference on Legal Knowledge and Information Systems (JURIX)*.
[13] Q. Betti, R. Khoury, S. Hallé, and B. Montreuil. 2019. Improving hyperconnected logistics with blockchains and smart contracts. *IT Professional* (2019).
[14] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. 2016. Formal Verification of Smart Contracts: Short Paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security (PLAS '16)*. Association for Computing Machinery, New York, NY, USA, 91–96. https://doi.org/10.1145/2993600.2993611
[15] T. Brandstätter, S. Schulte, J. Cito, and M. Borkowski. 2020. Characterizing efficiency optimizations in solidity smart contracts. In *IEEE Blockchain*.
[16] H. Chen, L. Lu, B. Massey, Y. Wang, and B. T. Loo. 2024. Verifying Declarative Smart Contracts. In *International Conference on Software Engineering (ICSE)*.
[17] H. Chen, G. Whitters, J. Amiri, M, Y. Wang, and B.T. Loo. 2022. Declarative smart contracts. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.
[18] T. Chen, Z. Li, H. Zhou, J. Chen, X. Luo, X. Li, and X. Zhang. 2018. Towards saving money in using smart contracts. In *International Conference on Software Engineering: New Ideas and Emerging Results (ICSE/NIER)*.
[19] Y. Chen, Y. Wang, M. Goyal, J. Dong, Y. Feng, and I. Dillig. 2022. Synthesis-powered optimization of smart contracts via data type refactoring. *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)* (2022).
[20] R. Chirkova and J. Yang. 2012. Materialized Views. *Foundations and Trends® in Databases* (2012).
[21] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 530–541. https://doi.org/10.1145/3377811.3380364
[22] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A Static Analysis Framework for Smart Contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. 8–15. https://doi.org/10.1109/WETSEB.2019.00008
[23] G. Governatori, F. Idelberger, Z. Milosevic, R. Riveret, G. Sartor, and X. Xu. 2018. On legal contracts, imperative and declarative smart contracts, and blockchain systems. *Artificial Intelligence and Law* (2018).
[24] A. Y. Halevy. 2001. Answering Queries Using Views: A Survey. *The VLDB Journal* (2001).
[25] Bo Jiang, Ye Liu, and Wing Kwong Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. 259–269.
[26] H. Jordan, B. Scholz, and P. Subotić. 2016. Soufflé: On synthesis of program analyzers. In *International Conference on Computer Aided Verification (CAV)*.
[27] A. Khatoon. 2020. A blockchain-based smart contract system for healthcare management. *Electronics* (2020).
[28] B.T. Loo, T. Condie, J.M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. 2005. Implementing declarative overlays. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*.
[29] I. Mami and Z. Bellahsene. 2012. A Survey of View Selection Methods. *SIGMOD Rec.* (2012).
[30] Julian Nagele and Maria A Schett. 2020. Blockchain Superoptimizer. arXiv:2005.05912 [cs.LO]
[31] K. Purnell and R Schwitter. 2022. *Towards declarative smart contracts*. Ph.D. Dissertation. Macquarie University.
[32] Meng Ren, Fuchen Ma, Zijing Yin, Ying Fu, Huizhong Li, Wanli Chang, and Yu Jiang. 2021. Making smart contract development more secure and easier. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1360–1370.
[33] Kenneth A Ross, Divesh Srivastava, and S Sudarshan. 1996. Materialized view maintenance and integrity constraint checking: Trading space for time. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*. 447–458.
[34] L. Ryzhyk and M. Budiu. 2019. Differential Datalog. *Datalog* 2 (2019), 4–5.
[35] F. Schär. 2021. Decentralized finance: On blockchain-and smart contract-based financial markets. *FRB of St. Louis Review* (2021).
[36] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. 2021. SmarTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution. In *30th USENIX Security Symposium (USENIX Security 21)*. 1361–1378.
[37] Dimitri Theodoratos, Timos Sellis, et al. 1997. Data warehouse configuration. In *VLDB*, Vol. 97. 126–135.
[38] Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. 2021. A Survey of Smart Contract Formal Specification and Verification. *ACM Comput. Surv.* 54, 7, Article 148 (jul 2021), 38 pages. https://doi.org/10.1145/3464421
[39] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. 67–82. https://doi.org/10.1145/3243734.3243780
[40] G. Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* (2014).
[41] Pengcheng Zhang, Feng Xiao, and Xiapu Luo. 2020. A Framework and DataSet for Bugs in Ethereum Smart Contracts. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 139–150. https://doi.org/10.1109/ICSME46990.2020.00023