EVA: A Symbolic Approach to Accelerating Exploratory Video Analytics with Materialized Views

Zhuangdi Xu*
Georgia Institute of Technology
xzdandy@gatech.edu

Gaurav Tarlok Kakkar*
Georgia Institute of Technology
gkakkar7@gatech.edu

Joy Arulraj
Georgia Institute of Technology
arulraj@gatech.edu

Umakishore Ramachandran
Georgia Institute of Technology
rama@cc.gatech.edu

ABSTRACT

Advances in deep learning have led to a resurgence of interest in video analytics. In an exploratory video analytics pipeline, a data scientist often starts by searching for a global trend and then iteratively refines the query until they identify the desired local trend. These queries tend to have overlapping computation and often differ in their predicates. However, these predicates are computationally expensive to evaluate since they contain user-defined functions (UDFs) that wrap around deep learning models.

In this paper, we present EVA, a video database management system (VDBMS) that automatically materializes and reuses the results of expensive UDFs to facilitate faster exploratory data analysis. It differs from the state-of-the-art (SOTA) reuse algorithms in traditional DBMSs in three ways. First, it focuses on reusing the results of UDFs as opposed to those of sub-plans. Second, it takes a symbolic approach to analyze predicates and identify the degree of overlap between queries. Third, it factors reuse into UDF evaluation cost and uses the updated cost function in critical query optimization decisions like predicate reordering and model selection. Our empirical analysis of EVA demonstrates that it accelerates exploratory video analytics workloads by 4× with a negligible storage overhead (1.001×). We demonstrate that the reuse algorithm in EVA complements the specialized filters adopted in SOTA VDBMSs.

CCS CONCEPTS

• **Computing methodologies** → **Image processing**; **Symbolic and algebraic manipulation**; • **Information systems** → **Query optimization**.

KEYWORDS

Video Analytics, Symbolic Computation, Database Management System

ACM Reference Format:
Zhuangdi Xu, Gaurav Tarlok Kakkar, Joy Arulraj, and Umakishore Ramachandran. 2022. EVA: A Symbolic Approach to Accelerating Exploratory Video Analytics with Materialized Views. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3514221.3526142

*Both authors contributed equally to the paper.

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9249-5/22/06.
https://doi.org/10.1145/3514221.3526142

1 INTRODUCTION

Researchers have presented novel techniques for efficiently analyzing visual data at scale in video database management systems (VDBMSs), including sampling, filtering, and specialized neural networks [35, 36, 44]. However, in *exploratory* video analytics, queries often exhibit a significant overlap of computation due to redundant execution of user-defined functions (*abbrev.*, UDFs) associated with computer vision tasks (*e.g.*, object detection). Prior efforts in video analytics have not extensively studied the problem of materializing and reusing the results of expensive UDFs. While there have been efforts in traditional database management systems (DBMSs) to handle expensive UDFs [13, 17, 19, 27, 45, 50], they do not leverage all of the opportunities present in video analytics.

Motivation. Consider a law enforcement officer analyzing a video data set for tracking a suspicious vehicle with the help of a witness. They typically first search for a global trend and then iteratively refine the query until they find the desired local trend [17]. During this query refinement process, queries tend to have overlapping computation. Initially, the witness may only recall the vehicle model (*e.g.*, SUV) and the approximate time-frame in which they saw the vehicle (*e.g.*, night time). So, the officer starts with Q_1 in Listing 1 that searches for all SUVs present during that time-frame to identify the suspicious vehicle. While going over the frames with SUVs returned by $Q1$, the witness might recall the color of the vehicle (*e.g.*, red). Then, the officer narrows down the search space and looks for the license plate of all red-colored SUVs ($Q2$). Lastly, in Q_3, the officer searches the entire dataset for the suspicious vehicle using the license plate information.

Multiple applications may query over the same video and their queries may also contain overlapping computation. For instance, a traffic planner may be interested in analyzing the traffic congestion over the entire day using Q_4. This task only requires a less-accurate (and faster) object detection model. Across these queries, several UDFs are redundantly evaluated (*i.e.*, VehicleModel, ObjectDetector, VehicleColor, Area, and License) on many frames. We seek to accelerate these queries by materializing and reusing the results of expensive UDFs, since these functions dominate the overall query processing time.

```
--- Q1: Suspicious Vehicle Tracking
SELECT timestamp, bbox, VEHICLE_COLOR(bbox, frame)
FROM VIDEO CROSS APPLY
OBJECT_DETECTOR(frame) ACCURACY 'HIGH'
WHERE timestamp > 6pm AND label = 'car'
AND AREA(bbox) > 0.3 AND
VEHICLE_MODEL(bbox, frame) = 'SUV';
--- Q2: Suspicious Vehicle Tracking
SELECT timestamp, bbox, LICENSE(bbox, frame)
FROM VIDEO CROSS APPLY
OBJECT_DETECTOR(frame) ACCURACY 'HIGH'
WHERE timestamp > 7pm AND timestamp < 8pm
AND label = 'car' AND AREA(bbox) > 0.3
AND VEHICLE_COLOR(bbox, frame) = 'red'
AND VEHICLE_MODEL(bbox, frame) = 'SUV';
-- Q3: Suspicious Vehicle Tracking
SELECT timestamp FROM VIDEO CROSS APPLY
OBJECT_DETECTOR(frame) ACCURACY 'HIGH'
WHERE timestamp> 4pm AND label= 'car' AND
AREA(bbox)>0.15 AND LICENSE(bbox, frame)= 'XYZ60';
--- Q4: Traffic Monitoring
SELECT timestamp, COUNT(*) FROM VIDEO CROSS APPLY
OBJECT_DETECTOR(frame) ACCURACY 'LOW' WHERE
label = 'car' AND AREA(bbox) > 0.15
GROUP BY timestamp;
```

Listing 1: Motivating Example — Illustration of overlapping computation in exploratory video analytics queries from two applications for: (1) suspicious vehicle tracking, and (2) traffic monitoring.

Manual Approach. The VDBMS could offload the burden of materializing the results to the application developers. With this approach, the developer manually decides to materialize the UDF results and then rewrites subsequent queries to reuse the materialized results. However, this approach suffers from two limitations. First, in exploratory video analytics, the data analyst often composes the subsequent query after examining the results of the current query. So they are not aware of reuse opportunities in advance. They will also need to manually refactor the query to leverage materialized views and determine whether the results of an UDF is worth materializing. This approach is error-prone and not suitable for complex queries. Second, the developer may not be aware of all the other applications running on the VDBMS. So, it is not possible to exploit opportunities for reusing results (*i.e.*, reuse opportunities) across applications. For example, the low-accuracy ObjectDetector in Q_4 of Listing 1 may reuse the results of high-accuracy ObjectDetector from Q_1 to Q_3, even though they are from different applications.

Challenges. We seek to automatically identify and leverage reuse opportunities in a VDBMS. However, there are three challenges with accomplishing this task.

I - Identifying Reuse Opportunities. Consider queries Q_1 and Q_2 in Listing 1. We notice that certain UDFs are present in both queries (*i.e.*, VehicleModel, VehicleColor, ObjectDetector, and Area), indicating an opportunity for reusing results. But, it is challenging to determine the degree of overlap between the predicates in Q_1 and Q_2 for each UDF due to the complexity of the expressions. Furthermore, the VDBMS must search for reuse opportunities across all previously executed queries (*i.e.*, not just compare two queries). For example, Q_3 in Listing 1 may reuse the results of License from Q_2 and ObjectDetector from both Q_1 and Q_2.

Reuse algorithms in traditional DBMSs [17, 19, 27, 33, 50, 55, 56, 64] rely on exact matching of sub-plans between two queries. This rigid approach does not handle the queries shown in Listing 1, as they vary in their complex predicates. Researchers have recently presented novel systems, such as Recycler [45] and HashStash [13], that extend query matching to compare different predicates. However, Recycler only supports a single range predicate, and HashStash only supports a few hard-coded rules for predicate analyses. We need a technique for capturing the *semantics* of these predicates and accurately determining the degree of overlap between them.

II - Automatically Rewriting the Queries. Second, a query may contain multiple UDFs, and each of them may differ in degree of reuse. For example, ObjectDetector in Q_3 may reuse the detection results for frames after 6 pm from Q_1. In contrast, License in Q_3 needs to be evaluated over a subset of vehicle bounding boxes from those frames, where Area (bbox) is between 0.15 and 0.3 (*i.e.*, relative to the frame size). In traditional DBMSs, rewriting queries to facilitate reuse involves substituting the sub-tree identified via query graph matching with a different sub-tree that reads from the materialized view. In contrast, reusing UDF results in queries shown in Listing 1 involves rewriting every UDF instance. The substitution technique used in traditional DBMSs works for sub-tree with root operator nodes (*e.g.*, projection and join operators). But it does not support selection operator with multiple UDF instances (*e.g.*, Q_2).

III - Reuse Impacts Cost of UDF Evaluation. Third, When the VDBMS replaces the UDF invocation with a view, it impacts several query optimization decisions (*e.g.*, predicate reordering and model selection). For example, the optimal evaluation order of VehicleModel and VehicleColor UDFs in Q_2 is to evaluate VehicleModel before VehicleColor. This is because the VDBMS can fully reuse their results from Q_1. If the VDBMS were to evaluate VehicleColor before VehicleModel, then it must unnecessarily evaluate the color of vehicles that are not SUVs (as there results are not computed in Q_1). Similarly, when the VDBMS selects a concrete model for the logical ObjectDetector in Q_4, even though a low-accuracy model would suffice for the traffic analysis application, it chooses to use the results of a high-accuracy model from the earlier queries in the suspicious vehicle tracking application.

Our Approach. In this paper, we present EVA, a VDBMS that automatically identifies opportunities for reusing UDF results to accelerate exploratory video analytics. EVA uses a novel technique for analysing UDF-based predicates using symbolic computing to determine the degree of reuse across queries for a given UDF. It employs a conditional apply operator to automatically transform UDF invocations to reuse materialized results from previous queries. Its ranking functions that guide predicate reordering and model selection decisions take the availability of reuse opportunities into consideration to determine the cost of UDF evaluation.

Contributions. We make the following contributions:

- We present a novel technique for determining how to reuse UDF results across queries using symbolic computing at query optimization time. The OPTIMIZER symbolically analyzes the predicates associated with every UDF invocation to identify opportunities for reusing results. We formulate transformation rules in EVA's Cascades-style OPTIMIZER to rewrite queries to leverage views.
- We propose a materialization-aware ranking function for reordering UDF-based predicates to accelerate queries, and provide the theoretical analysis. We reduce the model selection task to the weighted set cover problem and present a greedy algorithm that leverages symbolic computing to maximize reuse.
- We introduce a benchmark called VBENCH for evaluating the efficacy of reuse algorithms in exploratory video analytics. We develop one baseline, HASHSTASH [13], by tailoring the SOTA techniques for reusing results in traditional DBMSs to video analytics. We implement a second baseline around a tuple-level (*i.e.*, frame-level) function result caching scheme within EVA's EXECUTION ENGINE. We show that EVA outperforms these baselines on workloads with both low- and high-reuse potential. We also illustrate that the reuse algorithm used in EVA is complimentary to the widely-used filtering technique for accelerating video analytics.

2 BACKGROUND

In traditional DBMSs, the OPTIMIZER transforms the given SQL query into a query plan tree whose internal nodes are relational algebraic operators (*e.g.*, projection and selection). So, these systems reduce the problem of caching and reusing intermediate results to a sub-tree (*i.e.*, part of the plan tree) matching problem [50]. This approach suffers from two limitations in VDBMSs. First, UDFs are extensively used for exploratory video analytics. So, it is not sufficient to only identify reuse opportunities where sub-trees of two query plan trees are identical. As illustrated in Listing 1, even when the trees are distinct, the VDBMS should reuse results across overlapping UDF computations. Second, this approach does not capture reuse opportunities within an operator as it operates at the level of operators (*i.e.*, tree nodes). When the selection operator contains multiple expensive UDF-based predicates (*e.g.*, Q_2), all of these intra-operator predicates are candidates for reuse.
To address these limitations, EVA leverages a novel technique for better identifying reuse opportunities using symbolic computing (elaborated in §3).

SYMBOLIC COMPUTING. Symbolic computing [60] focuses on algorithmic manipulation of objects. Except for constants and variables, every arithmetic expression is considered as the symbol of an operator followed by a sequence of operands. Researchers have designed computer algebra systems that are capable of symbolic computation (*e.g.*, Sympy [43], Mathematica [29]).
EVA uses symbolic computing to analyze and simplify complex predicates. For example, it reduces "timestamp > 6pm OR timestamp > 9pm" to "timestamp > 6pm". We discuss how EVA uses symbolic computing to analyze complex predicates in § 4.1.

APPLY OPERATOR. In relational algebra, the APPLY operator is used to formulate correlated execution of sub-queries [15, 18]. It takes a relational input R and a parameterized expression $E(r)$, and evaluates the expression $E(r)$ for each row $r \in R$ and emits the tuples obtained by joining r and $E(r)$. Formally, it is defined as [18]:

$$R \, \mathcal{A}^{\otimes} \, E = \bigcup_{r \in R} (\{r\} \otimes E(r)) \tag{1}$$

where $\otimes$ is the join type (*e.g.*, inner join, cross join, *e.t.c.*). The conditional APPLY operator ($\mathcal{A}[p^*]$)[15] is an extension of the APPLY operator. It mimics an *if else* clause. It has a pass-through predicate p^* that acts as a guard predicate and only evaluates the parameterized expression $E(r)$ Eq. (1) if the guard is `TRUE`. EVA utilizes the APPLY operator and the conditional APPLY operator to rewrite queries to replace UDF invocations with materialized results.

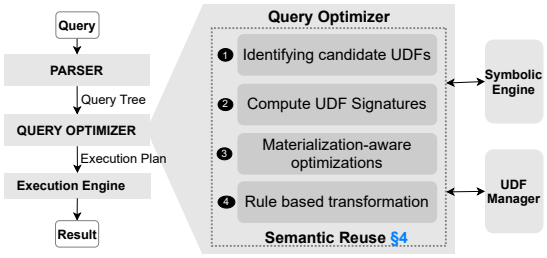


Figure 1: Overview of EVA– It takes in a client query, parses it and generates an optimized plan which is executed to generate results. We modify the OPTIMIZER to incorporate the reuse algorithm that accelerates query processing by leveraging the results of earlier UDF invocations.

PREDICATE REORDERING. The predicate ordering problem has been widely studied in traditional database systems [8, 26, 27]. The DBMS seeks to answer a query with an arbitrary number of conjunctions of restrictive predicates. These predicates are boolean-valued expressions that may invoke expensive UDFs. The DBMS must find a suitable ordering of these predicates that minimizes query processing cost.
Traditional DBMSs tackle this problem by computing a *rank* for each of the predicates using a *ranking function*. Then, the predicates are evaluated in ascending order based on the computed ranks. Eq. (2) presents the ranking function [26], where s is the selectivity and c is the per-tuple evaluation cost of the predicate.

$$r = \frac{s-1}{c} \tag{2}$$

Given that the selectivity s ranges between $[0, 1]$, the ranking function results in a negative value. Therefore, the smaller the rank, the better it is to evaluate the predicate. Intuitively, it is prioritizing the evaluation of inexpensive, highly selective predicates. However, it does not consider scenarios wherein EVA may already contain partial or fully materialized results for the predicates.

3 SYSTEM OVERVIEW

In this section, we first provide an overview of the *semantic reuse* algorithm used by EVA in § 3.1. We then discuss how this enables EVA to overcome the challenges in § 3.2. We conclude with a description of how EVA allows users to define UDFs in § 3.3.

3.1 Semantic Reuse Algorithm

EVA leverages the materialized UDF results of previous queries to accelerate subsequent queries. We design a *semantic reuse* optimization algorithm that is triggered after the canonical optimization algorithms have been applied. The key steps of this algorithm are shown in Fig. 1. The lifecycle of a query is as follows: The query is first processed by the parser that generates a parse tree. The OPTIMIZER takes in the parse tree and applies rule- and cost-based optimization techniques to generate a physical plan. The OPTIMIZER is based on the Cascades extensible query optimization framework [20, 63]. Finally, the EXECUTION ENGINE executes the given physical plan and returns the results to the client.

The semantic reuse algorithm leverages the *SymbolicEngine* to detect the degree of reuse across queries. It uses the information related to earlier UDF invocations within the *UDFManager* to facilitate reuse.

❶ **Identifying candidate UDFs.** For all the UDFs found in the query plan, the OPTIMIZER identifies the UDFs whose results are worth materializing. It uses the profiled evaluation cost to filters out inexpensive UDFs like AREA.

❷ **Compute UDF Signature.** A UDF's signature serves as a unique fingerprint and helps identify occurrences of the same UDF across queries. The signature S_u of a UDF u is defined as a tuple $S_u = [N_u; I_u]$ where:

- N_u is the name of the UDF u
- I_u is the set of source tables or views or UDFs that EVA must access for evaluating UDF u.

When the OPTIMIZER applies the canonical transformation rules for rewriting the query, it keeps track of the signature associated with every UDF occurrence within the query and appends it to the UDFMANAGER. EVA reuses results across UDFs with identical signatures. The UDFMANAGER maintains a mapping from the UDF signature to the corresponding materialized view.

❸ **Materialization-aware optimizations.** Given a candidate UDF and its historical invocations from the UDFMANAGER, the OPTIMIZER seeks to answer two questions. First, if multiple UDFs must be evaluated on the same input table, what is the optimal ordering in which these UDFs should be evaluated? Second, if a collection of deep learning models are suitable for a given vision task, what are the appropriate models (physical UDFs) to minimize the execution cost? In § 4.2 and § 4.3, we discuss how the OPTIMIZER leverages the SYMBOLICENGINE to answer these questions.

❹ **Rule based transformation on the candidate UDFs.** Lastly, the OPTIMIZER performs a rule-based transformation on the candidate UDFs to leverage existing results in materialized views. We present this transformation step in § 4.4.

3.2 Solution Overview

EVA addresses the challenges outlined in § 1 as follows:

I - Identifying Reuse Opportunities. To identify the overlapping computation between UDF invocations X and Y with the same signature, EVA utilizes symbolic computing to compute three *derived predicates*: (1) intersection, (2) difference, and (3) union of predicates in X and Y. Intersection predicate denotes input tuples (*i.e.*, frames) where the latter UDF invocation (*i.e.*, Y) may reuse the results from the former invocation (*i.e.*, X). Difference predicate represents input tuples where the reuse is not feasible and Y must be evaluated. Union predicate denotes input tuples where the materialized results are available after both X and Y are evaluated. It uses these derived predicates to detect reuse opportunities.

```
CREATE [OR REPLACE] UDF YOLO
INPUT = (frame NDARRAY UINT8(3, ANYDIM, ANYDIM))
OUTPUT = (labels NDARRAY STR(ANYDIM), bboxes
  NDARRAY FLOAT32(ANYDIM, 4))
IMPL = 'udfs/yolo.py'
LOGICAL_TYPE = ObjectDetector
PROPERTIES=('ACCURACY'='HIGH')
```

Listing 2: Defining a UDF– This statement creates a YOLO object detection UDF and specifies the accuracy property.

II- Reuse Impacts Cost of UDF Evaluation. We formulate the cost of an UDF invocation to consider the availability of materialized results by using the selectivity of the intersection predicate and difference predicate obtained from the SYMBOLICENGINE. The OPTIMIZER uses this updated cost function in the predicate reordering task to compose a materialization-aware ranking function. When the UDF-based predicate transformation rule unpacks a selection operator containing multiple UDF-based predicates, the order is determined by this new ranking function. Similarly, in the model selection task, the OPTIMIZER maps it to a weighted set cover problem. The weights are defined by using the selectivity of the intersection predicate.

III - Rewriting the Queries. We introduce two transformation rules in the OPTIMIZER for facilitating reuse: (1) an UDF-based predicate transformation rule for unpacking a selection operator that contains multiple UDF invocations, and (2) a materialization-aware transformation rule that fully (or partially) replaces the expensive UDF invocation with the materialized results of previous queries and injects a store operator for materializing the remaining UDF results. Both rules are based on the conditional apply operator [15]. They enable effective reuse of UDF results regardless of the location of UDF in the query (*e.g.*, attribute list in projection operator or predicate in selection operator).

3.3 Defining UDFs

EVA supports a declarative SQL-like query language called EVA-QL. It allows users to invoke deep learning models in the form of UDFs. Listing 2 presents an example where the user defines a UDF wrapping around the YOLO object detection model. The user may utilize this UDF in subsequently queries to detect objects, as illustrated in Listing 1.

In Listing 2, the user specifies the input(s) and output(s) of the UDF. For instance, YOLO consumes a video frame of arbitrary dimensions and produces labels and bounding boxes of detected objects. `IMPL` specifies the path to the implementation class for the UDF. `LOGICAL_TYPE` specifies the model type of the UDF (*e.g.*, ObjectDetector). The user also specifies the expected accuracy in `PROPERTIES`. The OPTIMIZER uses these properties while picking physical models for a logical vision task (elaborated in § 4.3).

Modular vs Monolithic UDFs. EVA allows users to create arbitrary UDFs. For instance, in Listing 1, they may create a specialized UDF that only detects red SUVs and use it in Q_2. We refer to

such specialized UDFs as monolithic UDFs. EVA will reuse results if the same monolithic UDF is invoked again during exploratory analysis. However, using separate, modular UDFs (*e.g.*, VehicleColor for detecting the color of the vehicle and VehicleModel for detecting the model of the vehicle) allow users to flexibly combine them (*e.g.*, the analyst may use these UDFs for detecting red sedans or blue SUVs later). EVA supports reuse with both modular and monolithic UDFs.

4 SEMANTIC-REUSE ALGORITHM

In this section, we describe the components of the semantic reuse algorithm in more detail. We first discuss how EVA leverages symbolic computing in § 4.1. We then present how EVA rewrites queries in § 4.4. We next describe how EVA leverages materialized views in § 4.2. We conclude with a description of the logical UDF reuse optimization in § 4.3.

4.1 Symbolic Predicate Analysis

Symbolic computation focuses on algorithmic manipulation of mathematical expressions. Except for numbers and variables, mathematical expression may be viewed as the symbol of an operator followed by a sequence of operands [60]. Researchers have designed symbolic computation algorithms to process and simplify such mathematical expressions [29, 43]. EVA leverages symbolic algorithms to analyze and simplify complex predicates in queries.

Overview. The VDBMS must evaluates the given UDF over the subset of tuples that satisfy a predicate. For example, in Q1 (Listing 1), the predicate associated with ObjectDetector is: timestamp > $6pm$. The predicate associated with VehicleModel is: timestamp > $6pm \wedge$ label = 'car' $\wedge$ Area(bbox, frame) > 0.3.

EVA's optimizer leverages symbolic computing to analyze the predicates associated with the UDF invocations that share the same signature. Consider two UDF invocations u_1 and u_2 with the same signature. Let their predicates be p_1 and p_2, respectively. We define three fundamental *derived predicates* based on p_1 and p_2 to determine the overlapping computation between u_1 and u_2:

- Intersection: $\mathsf{INTER}(p_1, p_2) = p_1 \wedge p_2$
- Difference: $\mathsf{DIFF}(p_1, p_2) = (\neg p_1) \wedge p_2$
- Union: $\mathsf{UNION}(p_1, p_2) = p_1 \vee p_2$

Logically, the intersection predicate denotes the overlapping computation between u_1 and u_2. The difference predicate denotes the non-overlapping computation (*i.e.*, u_2 is computed but not u_1). The union predicate denotes the computations where either u_1 or u_2 is evaluated.

Leveraging Results of Symbolic Analysis. The UDFManager keeps maintaining the *aggregated* predicate $\tilde{p}_u$ for each unique UDF signature u. Formally, it is the union of all the predicates associated with UDF u across all the queries. It represents the tuples where the materialized results are available for UDF with signature u. When EVA encounters an UDF with signature u for the first time, since it has never executed u, $\tilde{p}_u$ is instantiated as False. When the optimizer receives a query containing UDF u with q as the associated predicate, it updates $\tilde{p}_u = \mathsf{UNION}(\tilde{p}_u, q)$.

For analyzing the reuse opportunity, it computes $\mathsf{INTER}(\tilde{p}_u, q)$ to symbolically identify the tuples it has over which it has computed u before and $\mathsf{DIFF}(\tilde{p}_u, q)$ for those over which it has not computed u. We abbreviate these three derived predicates: $\mathsf{INTER}(\tilde{p}_u, q)$ as $p_\cap$, $\mathsf{DIFF}(\tilde{p_u}, q)$ as p_-, and $\mathsf{UNION}(\tilde{p_u}, q)$ as $p_\cup$ in the rest of the paper.

Algorithm 1: Symbolic Predicate Analysis

```
Input  : predicate: input predicate
Output : simplifiedPredicate: simplified predicate
1  Procedure ReducePredicate(predicate)
2      ❶ DNFPredicate ← DNF (predicate)
3      foreach conjunctive in DNFPredicate do
4          ❷ ReduceConjunctive (conjunctive)
5      repeat
6          c1, c2 ← PopTwoConjunctives (DNFPredicate)
7          ❸ c1, c2 ← ReduceUnionConjunctives (c1, c2)
8          PushConjunctives (DNFPredicate, c1, c2)
9      until TimeOut or NoChange
10     return DNFPredicate
11 Procedure ReduceUnionConjunctives (c1, c2)
12     foreach dimension in c1 ∨ c2 do
13         fc1 ← FilterDimension (c1, dimension)
14         fc2 ← FilterDimension (c2, dimension)
15         if fc1 ⊃ fc2 or fc1 ⊂ fc2 then
16             return ReduceUnionSingleDimension (c1, c2, dimension)
17     return c1, c2
```

Predicate Syntax. EVA supports predicates with the following syntax:

$$\textbf{p} ::= \textbf{expr cp expr} \mid \textbf{p logic p} \mid \textbf{NOT p}$$
$$\textbf{cp} ::= > \mid < \mid = \mid \neq \mid \leq \mid \geq$$
$$\textbf{logic} ::= \textbf{AND} \mid \textbf{OR}$$

A predicate may be: (1) a comparison of two expressions (*e.g.*, columns, UDFs, constant values), (2) a combination of two predicates using a boolean operator, and (3) negation of another predicate. EVA seeks to reduce the number of atomic predicates (*i.e.*, a predicate that cannot be reduced further into simpler predicates) in the three derived predicates.

Challenge. The number of atomic predicates governs the complexity of the intersection, union, and difference operations. So, it is important to simplify the outcome of these operations. EVA leverages a computer algebra system to reduce monadic predicates (*e.g.*, $\mathsf{UNION}(5 < x \wedge x < 15, 10 < x \wedge x < 20) \rightarrow 5 < x \wedge x < 20$). However, it is challenging to reduce polyadic predicates[1] (*e.g.*, $\mathsf{UNION}(5 < x \wedge 10 < y, 10 < x \wedge 15 < y) \rightarrow 5 < x \wedge 10 < y$).

Algorithm. EVA uses Algorithm 1 to simplify a polyadic predicate. ❶ It converts the input predicate to disjunctive normal form (DNF). ❷ It independently reduces every conjunctive within the DNF. To reduce a multi-variable conjunctive predicate, EVA uses a computer algebra system. Next, it focuses on reducing the predicate across conjunctives. To do so, ❸ it repeatedly pops two conjunctives and checks whether it is possible to reduce their union predicate . It repeats this procedure until there remain no additional reduction opportunities between any pairs of conjunctives or the symbolic analysis goes beyond the allocated time budget.

Figure 2 illustrates three cases stemming from reducing the union of two conjunctive predicates involving two variables (*i.e.*, two-dimensional predicates).

[1] Minimizing propositional logic (boolean formula) is a hard problem [7]. Predicate logic is a superset of propositional logic.

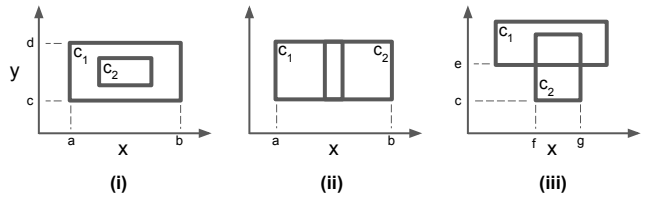


Figure 2: Reduction of Union of Conjunctives — Illustration of the two-dimensional cases where c_1 and c_2 involve two variables (x and y). A rectangle represents a predicate with four atomic terms. For example, c_1 in (i) is: $a < x \wedge x < b \wedge c < y \wedge y < d$.

- Case i — c_2 is a subset of c_1 in both x and y dimensions, so their union is c_1.
- Case ii — it is possible to concatenate the predicates along x dimension, so the union is $(a < x \wedge x < b) \wedge (c < y \wedge y < d)$.
- Case iii — it is possible to remove the overlapping region from the c_2 to make the conjunctives disjoint, so their union is $c_1 \vee (f < x \wedge x < g \wedge c < y \wedge y < e)$.

In all three cases, c_2 is a subset of (or equal to) c_1 in one dimension, [2] and the OPTIMIZER uses the computer algebra system to reduce the union of the other dimension. This technique generalizes to predicates that do not map to a rectangle. *ReduceUnionConjunctives* in Algorithm 1 extends the concept into N-dimensional predicates. Specifically, c_1 needs to be the subset of c_2 in at least $N-1$ dimensions, or the other way around, where N is the dimensionality of $c_1 \vee c_2$. The OPTIMIZER then reduces the union of the remaining dimension using the computer algebra system.

4.2 Materialization-Aware Optimization

Replacing expensive UDF invocations with the materialized results of previous queries reduces the cost of UDF evaluation. So we need to adjust the UDF cost based on the availability of views in optimization tasks such as predicate reordering and model selection (elaborated in § 4.3).

REDEFINING THE COST OF A UDF INVOCATION. As shown in Eq. (3), we compute the expected cost of evaluating a predicate o containing a UDF as a function of the cardinality of the input relation $|R|$. Let C_M be the cost of reading the materialized view associated with UDF in o. Let c_r be the per-tuple cost for reading a tuple from the input relation R. Let c_e be the per-tuple cost of evaluating the UDF. Let $s_{p\_}$ be the selectivity of the difference predicate $p\_$ calculated from symbolic predicate analysis.

$$T(o, |R|) = (3C_M + |R|c_r) + |R|s_{p\_}c_e = |R|(\frac{3C_M}{|R|} + c_r + s_{p\_}c_e) \quad (3)$$

Here, the first part is the expected cost of a join operation between R and M [38], and the second part is the expected cost of evaluating the UDF on the fraction of input tuples missing in the view. The join operator combines the input video and UDF's materialized view. For example, the object detector's materialized results are joined with the video table to read the overlapping outcomes of the object detection from previous queries. We will discuss how EVA rewrites the query plan to leverages the view in § 4.4.

When the materialized views and tables are on disk, we estimate the join cost to $3C_M$ [38]. During the build phase, EVA reads the table into memory, creates a hash table, and stores it back on disk if it cannot keep it in memory. Then during the probe phase, it reads the hash table from the disk. So, in the worst case, it leads to 3 IO operations. However, in practice, the join term is negligible compared to the cost of evaluating the UDF on the fraction of input tuples missing in the view (second term in Eq. (3)) and may be ignored (§5.3).

WHAT IS THE OPTIMAL ORDERING OF EVALUATING THE UDF-BASED PREDICATES?. In video analytics, an analyst may often use a compound predicate with multiple UDFs. For example, in Listing 1, Q_2 needs to evaluate two UDF invocations in the predicates: VEHICLECOLOR and VEHICLEMODEL. The order in which these UDFs are evaluated lowers the overall execution time of the query by 3–6× (§ 5.4).

Using the UDF cost function in Eq. (3), EVA adopts a novel ranking function, shown in Eq. (4), that takes materialized views into consideration.

$$r = \frac{s-1}{s_{p\_} \times c_e + c_r} \quad (4)$$

Here, s denotes the selectivity of the UDF-based predicate. EVA leverages existing histogram-based methods in traditional database systems to calculate the selectivity of predicates [30, 51]. Intuitively, Eq. (4) prioritizes highly selective predicates with lower evaluation costs. The key difference from the traditional ranking function (Eq. (2)) is that in the new formulation, the evaluation cost is proportional to the fraction of tuples not present in the materialized view ($s_{p\_}$). The c_r term comes from the join cost. However, in practice, it is negligible and can be ignored. For instance, when the views are stored on hard disk, the profiled values for the FASTERRCNNRESNET50 are $c_r = 1.8$ ms and $c_e = 99$ ms (§ 5). If the views are stored on another storage medium (*e.g.*, in memory), we need to update c_r accordingly. We next formally prove the correctness of the proposed ranking function.

THEOREM 4.1. *Let O be a conjunctive ordering of n boolean valued, independent* [3] *predicates, $o_1, o_2, ..., o_n$. The expected evaluation cost of O is minimal when the predicates are evaluated in ascending order of rank computed using the ranking function in Eq.* (4).

PROOF. We may construct any ordering of the predicates by a series of swaps between adjacent predicates $o_i o_j$ such that $i < j$ [24]. To prove that an ordering O is optimal, it is sufficient to show that any such arbitrary swap does not decrease the expected evaluation cost of O. Consider an arbitrary ordering O:

$$O : o_1, ..., o_k, o_{k+1}, ..., o_n$$

We define the expected evaluation cost $T(O, |R|)$:

$$T(O, |R|) = T(o_1, |R|) + T(o_2, s_1|R|) + ... + T(o_k, s_1...s_{k-1}|R|)$$
$$+ T(o_{k+1}, s_1...s_{k-1}s_k|R|) + ... + T(o_n, s_1...s_{n-1}|R|)$$

[2] EVA uses a computer algebra system to check whether $(\neg c_{1,d}) \wedge c_{2,d} ==$ False or $(\neg c_{2,d}) \wedge c_{1,d} ==$ False to determine the subset relationship for a given dimension. Here, $c_{i,d}$ is the conjunctive of atomic formulas involving dimension d in predicate c_i.

[3] The theorem holds under the assumption that the predicates are independent. While this assumption may not always hold in practice, theoretical analysis in predicate re-ordering literature [8, 26] and video analytics systems [37, 40] also make this assumption to simplify the analysis.

We derive $T(o_k)$ by substituting $|R|$ with $s_1 s_2 ... s_{k-1}|R|$, where s_i denotes the selectivity of predicate o_i [24, 27]. This is because the size of input relation reduces to the application of earlier predicates in O. Consider another ordering $O^{'}$ by interchanging o_k and o_{k+1}:

$$O^{'} : o_1, ..., o_{k+1}, o_k, ..., o_n$$

$T(O^{'})$ will be similar to $T(O)$ expect for $T(o_k)$ and $T(o_{k+1})$. For ease of presentation, let $k = 1$. A similar argument holds for any arbitrary value of k.

$$\begin{aligned}
&T(O^{'}, |R|) - T(O, |R|) \\
&= |R|(\frac{3C_{2M}}{|R|} + c_r + s_{2p_}c_{2e}) + s_2|R|(\frac{3C_{1M}}{s_2|R|} + c_r + s_{1p_}c_{1e}) \\
&- |R|(\frac{3C_{1M}}{|R|} + c_r + s_{1p_}c_{1e}) - s_1|R|(\frac{3C_{2M}}{s_1|R|} + c_r + s_{2p_}c_{2e}) \\
&= |R|((s_2 - 1) \times s_{1p_}c_{1e} - (s_1 - 1) \times s_{2p_}c_{2e} + (s_2 - s_1) \times c_r) \\
&= |R|(s_{2p_}c_{2e} + c_r)(s_{1p_}c_{1e} + c_r)(\frac{s_2 - 1}{s_{2p_}c_{2e} + c_r} - \frac{s_1 - 1}{s_{1p_}c_{1e} + c_r}) \\
&\geq 0
\end{aligned}$$

The last expression in the closed parenthesis is positive since we order the predicates using the ranking function Eq. (4). □

4.3 Logical UDF Reuse

We next present a technique of reusing UDF results in the optimizer based on their logical semantics. Consider Q1 searching for a suspicious vehicle in Listing 1. The officer may tolerate different physical implementations of the logical UDF, ObjectDetector provided they meet the accuracy requirements (similar to logical and physical operators in query optimization [20]). The physical UDF may be (1) yolo-tiny [2], (2) FasterRCNNResnet50, or (3) FasterRCNNResnet101 [54]. This scenario also arises across applications. For example, the traffic monitoring application might also be running an object detection model, which presents the opportunity to reuse results. These physical UDFs may vary in their accuracy, inference time, and availability of materialized results. EVA considers the materialized views of *all* the models and automatically substitutes the logical vision task with one or more physical models. We reduce the problem of selecting the optimal physical UDFs that minimize the execution cost to the weighted set cover problem.

Weighted Set Cover Problem. The weighted set cover problem is defined as follows. Given a universe U ($|U| = n$) and a collection of sets $S = \{S_1, S_2, ..., S_m\}$, $S_i \subseteq U \ \forall i$. Each set S_i has a weight $w_i \geq 0$. The set cover is a subset $I = \{1, 2, 3, ..., r\}$ such that $\cup_{i \in I} S_i = U$. The weighted set cover problem finds a set cover with the minimum overall weight $\sum_{i \in I} w_i$. The weighted set cover problem is NP-Complete [10].

We use a polynomial-time greedy algorithm to solve this problem. Suppose $Uncovered \subseteq U$ is set of uncovered elements of the universe U. At each iteration i, the algorithm picks the set S_i that minimizes $\frac{w_i}{|S_i \cap Uncovered|}$. The idea is to minimize the cost per uncovered element at each iteration. It achieves a $\ln n - approximation$ [10] and is the best possible approximation for any polynomial algorithm [16, 41].

Optimal set of physical UDFs. We prove that selection of the optimal set of physical UDFs reduces to a weighted set cover problem in Theorem 4.2. Consider the tuples in materialized view m_i of the physical UDFs x_i as a set S_i in collection S. The cost of reading the materialized view $C(m_i)$ is mapped to weight w_i. During each iteration, the greedy algorithm picks the physical UDF that minimizes $\frac{C(m_i)}{|m_i \cap Uncovered|}$. To compute the denominator (*i.e.*, the cardinality of the uncovered elements), we leverage the symbolic engine. Specifically, we use the selectivity of the intersection predicate. We provide more details below.

Algorithm 2: Logical UDF Reuse – The algorithm to rewrite the logical UDF with the corresponding physical UDFs that minimizes the execution cost.

Input : sig: UDF signature, $\mathcal{C}$: set of UDF constraints, q: assocaited predicate for UDF
Output: $\mathcal{D}$: optimal set of equivalent physical UDFs

1 **Procedure** OptimalPhysicalUDFs(sig, $\mathcal{C}$, q)
2 $X \leftarrow$ PhysicalUDFs(sig, $\mathcal{C}$) ▷ Phy UDFs that satisfy constraints
3 $y \leftarrow arg\,min_{x \in X}\ c_x$ ▷ Min cost UDF
4 **repeat**
5 **for** $x \in X$ **do**
6 Compute $W(x, q) = \frac{C(m_X)}{s_{p \cap_X}|m_X|}$ ▷ Cost per uncovered tuple
7 $x^* \leftarrow argmin_{x \in X} W(x, q)$ ▷ UDF with min W(x, q)
8 **if** $W(x^*, q) < c_y$ **then**
9 $\mathcal{D} \leftarrow \mathcal{D} \cup \{(x^*, p_{\cap_{x^*}})\}$ ▷ Select mat view of x^*
10 $q \leftarrow \text{DIFF}(\tilde{p}_{x^*}, q)$
11 **else**
12 $\mathcal{D} \leftarrow \mathcal{D} \cup \{(y, q)\}$ ▷ Select cheapest UDF
13 $q \leftarrow \emptyset$
14 **until** $q \neq \emptyset$
15 **return** $\mathcal{D}$

Algorithm. The optimizer uses Algorithm 2 to find the optimal set of physical UDFs. It first retrieves the set of physical UDFs X, satisfying the required constraints (Line 2). Next, it finds the cheapest UDF y with cost c_y in Line 3, which will be used when no materialized view is picked. Then, it computes the cost per uncovered tuple (Line 6). The intersection predicate (§ 4.1) selects a subset of tuples in the materialized view that satisfies the predicate q. So, the selectivity of the intersection predicate is used to calculate the cardinality of uncovered tuples $s_{p \cap_x}|m_x|$. In Line 8, it checks if it is beneficial to pick a materialized view or instead run the cheapest UDF. Specifically, if the cost per tuple of the materialized view is lower than running the cheapest UDF, it selects the materialized view (Lines 9 to 10). It updates the query predicate by computing the difference between the UDF's predicate ($\tilde{p}_{u^*}$) and the query predicate. Otherwise, it decides to run the cheapest UDF for the remaining range (Lines 11 to 13). In § 5.4, we show that the logical UDF reuse optimization delivers a 2.2× workload speedup.

Theoretical Analysis. We next provide an analysis of the problem reduction.

Theorem 4.2. *Given the set of physical UDFs $X = \{x_1, x_2..., x_k\}$, the corresponding materialized views $M = \{m_1, ..., m_k\}$ and the associated predicate q, the problem of selecting the optimal set of physical UDFs $Y \subseteq X$ reduces to a weighted set cover problem.*

Proof sketch. Suppose W be the set of all possible tuples, $C(m_i)$ the cost of reading the materialized view m_i, and c_i the execution cost of UDF x_i. The universe $U(|U| = n)$ is the set of tuples over which the UDF needs to be evaluated. They satisfy the associated predicate q.

$$U = \sigma_q(W)$$

Given the cost of the cheapest UDF c_j ($j = argmin\ c_j$), $p_{\cap_{x_i}}$ = INTER($\tilde{p}_{x_i}, q$) (§ 4.1), and $P(q), q \in [1, 2^n]$ is the q^{th} element in the power set of U (ordered arbitrarily). The collection of set S and the corresponding weights are defined as follows.

$$S = \{S_1, ..., S_k, S_{k+1}, ..., S_{k+2^{|U|}}\} \quad \text{such that}$$

$$S_i = \begin{cases} \sigma_{p_{\cap_{x_i}}}(U), & 1 \le i \le k \\ P(i-k), & k+1 \le i \le k+2^{|U|} \end{cases} \tag{5}$$

$$w_i = \begin{cases} C(m_i), & 1 \le i \le k \\ c_j|P(i-k)|, & k+1 \le i \le k+2^{|U|} \end{cases} \tag{6}$$

$S_i, i \in [1, k]$ represents the subset of tuples in the materialized view m_i that satisfy the intersection predicates q. $S_i, i \in [k+1, k+2^{|U|}]$ represents all the subsets of the tuples on which the UDF needs to be executed (all subset of U). We append the power set to handle two scenarios: (1) the tuples for which we do not have the materialized results, and (2) the scenario where using the cheapest UDF is better than reading results from the view. The weight $w_i, i \in [1, k]$ is the cost of reading the materialized view. For elements in the power set, weight $w_i, i \in [k+1, k+2^{|U|}]$ is the cost of executing the cheapest UDF on the tuples in the set. As the assigned weights w_is are proportional to the execution cost, the optimal weighted set cover for the above problem gives the optimal set of physical UDFs that minimizes the execution cost.

4.4 Rule-Based Query Rewrite

We design two transformation rules to rewrite UDF invocations in the query plan:

I - UDF-Based Predicate Transformation Rule. The optimizer leverages the apply ($\mathcal{A}$) operator to transform UDF invocations into relational operators. Fig. 3 illustrates this rule-based transformation. Suppose the selection operator contains multiple UDF-based predicates. In that case, the optimizer first reorders the predicates based on the materialization-aware ranking (elaborated in § 4.2), then chains their transformation using the apply operator (*i.e.*, the output of the preceding UDF-based predicate is the input of the succeeding one).

II - Materialization-Aware Transformation Rule. A straightforward transformation to reuse the materialized results consists of: (1) using the intersection predicate ($p_\cap$) to filter out the materialized results, (2) using the difference predicate (p_-) to filter out the input relation and evaluate the UDF on the remaining tuples, and (3) using the union operator to aggregate the results. However, this solution has two drawbacks. First, $p_\cap$ and p_- may be complex, and evaluating them over every tuple will be expensive. Second, $p_\cap$ and p_- may contain other UDFs, and the optimizer needs to consider reusing their materialized results. So, this approach requires recursion.

To circumvent this problem, EVA's materialization-aware transformation rule entails three modifications, as shown in Fig. 4:

❶ If the optimizer finds out that there exists a materialized view (M) for the UDF signature u, it introduces a `LEFT OUTER JOIN` operator that operates on the input relation R and M.

❷ It replaces the apply operator with a *conditional apply* operator. Notice that for tuples that are missing in view M, the output columns are populated with `NULL`s. So the pass-through predicate p^* of the *conditional apply* operator guarantees that the VDBMS only evaluates the UDF for tuples with missing values.

❸ It introduces a new `STORE` operator on top of the UDF, which ensures that the VDBMS appends the UDF evaluation results to the materialized view M_u, that may be reused for processing future queries.

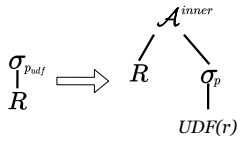


Figure 3: UDF-Based Predicate Transformation Rule

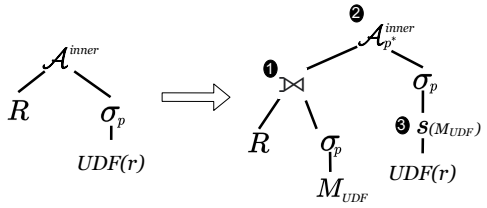


Figure 4: Materialization-Aware Transformation Rule

If the intersection predicate ($p_\cap$) or the different predicate (p_-) is `FALSE` (after symbolic reduction), we may simplify this transformation accordingly. For example, if $p_\cap$ is `FALSE`, this implies that M_u does not contain any results of the required UDF invocation, so the optimizer skips the `LEFT OUTER JOIN` operator (❶) in the transformation. If p_- is `FALSE`, then all the results are available in view M_u. In this case, in Fig. 4, the optimizer removes the apply operator (❷) and its right subtree because the join operator (❶) provides all the required results.

5 EVALUATION

In our evaluation, we seek to answer the following questions:

- How does the UDF-centric reuse algorithm in EVA compare against reuse algorithms in traditional DBMSs and canonical function caching technique (§ 5.2)?
- What is the overhead of reuse operations (§ 5.3)?
- How effective is EVA's semantic reuse algorithm, including symbolic predicate reduction, materialization-aware predicate reordering, and logical UDF reuse (§ 5.4)?
- What is the impact of video length and content on EVA's reuse algorithm (§ 5.5)?
- How does EVA's reuse algorithm complement the specialized filters used in SOTA VDBMSs (§ 5.6)?

5.1 Experimental Setup

Implementation. We implement EVA in Python. We use Antlr [48] to parse the input query and generate the parse tree. We manage the catalog in a traditional DBMS using SQLAlchemy [6]. We implement the storage engine using the Petastorm library [21]. It stores the videos on disk using the Apache Parquet format. The

Table 1: Illustrative queries in VBENCH-HIGH

SELECT <> FROM VIDEO CROSS APPLY FastRCNNObjectDetector(frame)	
Q1 id < 10000 ∧ label = 'car' ∧ area > 0.3 ∧ CarType(frame, bbox) = 'Nissan';	**Q2 - Zoom out:** id < 10000 ∧ label = 'car' ∧ CarType(frame, bbox) = 'Nissan';
Q3 - Zoom in: id < 10000 ∧ area > 0.25 ∧ label = 'car' ∧ CarType(frame, bbox) = 'Nissan' ∧ ColorDet(frame, bbox) = 'Gray';	**Q6 - Shifting:** id > 7500 ∧ label = 'car' ∧ ColorDet(frame, bbox) = 'Gray';

EXECUTION ENGINE consumes this data after converting it to a Pandas Dataframe [42], and relies on the Pytorch framework [49] to evaluate deep learning-based UDFs in the queries. EVA currently contains 28 K lines of code. The SYMBOLICENGINE uses the SymPy library [43] for the symbolic analysis of predicates to guide reuse decisions. The developer may extend the Cascades-style OPTIMIZER by adding additional rewrite rules over time. The system is available at: https://github.com/georgia-tech-db/eva.

WORKLOAD GENERATION. There are no standard benchmarks for exploratory video analytics. To test EVA, we develop the VBENCH benchmark that captures a wide range of queries and stress tests EVA's capability to reuse results.

VIDEO DATASETS. We evaluate EVA on two datasets.

- UA-DETRAC [59]. To study the impact of video length, we construct three video sets based on UA-DETRAC: SHORT-UA-DETRAC (5 clips with 7.5k frames in total), MEDIUM-UA-DETRAC (10 clips with 14k frames in total), and LONG-UA-DETRAC (20 clips with 28k frames in total).
- JACKSON (night-street from [35]) with 14k frames. JACKSON has a lower resolution (600 × 400) compared to UA-DETRAC (960 × 540). Furthermore, on average, JACKSON has fewer vehicle appearances (0.1 vehicle per frame) than UA-DETRAC (8.3 vehicles per frame). We adjust the number of frames to be consistent with that of MEDIUM-UA-DETRAC.

QUERY SETS AND WORKLOAD GENERATION. We construct two query sets with (1) low-, and (2) high-reuse potential denoted by VBENCH-LOW, and VBENCH-HIGH, respectively. Here are the key properties of these query sets:

- VBENCH-LOW: The average overlap of frames read from the video dataset by two subsequent queries is 4.5%. This represents a scenario where the analyst skims through different parts of the video.
- VBENCH-HIGH: The average overlap is 50%. This represents a scenario where the analyst iteratively refines the query over a particular part of the video.

Every query set contains 8 queries focusing on vehicles in the videos (similar to the motivating example in §1). Every query contains an APPLY operator to connect the video with the object detection UDF. The queries have up-to five predicate clauses, where three of them are direct-column predicates (*i.e.*, id, label, and scores), and two of them are UDF-based predicates (*i.e.*, vehicle color, and type).
Table 1 lists the illustrative queries. FASTERRCNNRESNET50 [54], CARTYPE, and COLORDET are UDFs for object detection, vehicle type recognition, and color classification, respectively. Since they are expensive, EVA identifies them as candidate UDFs for reuse. The queries emulate an exploratory analysis for a suspicious vehicle. Typically, such analysis is a combination of zooming in/out and range shifting operations [13, 58]. The user begins by looking for a car, likely a Nissan (Q1). Based on the result of Q1, they relax the constraint on the area of the bounding box (Q2: Zooming out). Next, they add the color constraint to refine the search further (Q3: Zooming in). In subsequent queries, they shift the frame range of the query (Q6: Shifting range). We evaluate every workload from a clean state (*i.e.*, no available materialized results). Unless otherwise specified, the experiments are based on the MEDIUM-UA-DETRAC video dataset and VBENCH-HIGH query set.

Table 2: Hit Percentage

Hit Percentage (%)	HASHSTASH	FUNCACHE	EVA
VBENCH-LOW	2.02	24.68	24.68
VBENCH-HIGH	5.62	66.01	66.01

BASELINES. We reimplement the key ideas of HASHSTASH and function caching technique within EVA for a fair comparison. We refer to these baselines as HASHSTASH and FUNCACHE.

- HASHSTASH: It utilizes a recycler graph to keep track of the plans associated with previously executed queries. Every node in the recycler graph represents an operator in the plan (*e.g.*, hash-join and hash-aggregate). It materializes the results of the operator. To exploit reuse opportunities, it first does a sub-tree matching between the query and the recycler graph without requiring predicates to be identical. It then deduplicates the union of materialized results of all matched operators and applies the query's predicates to answer the query.
- FUNCACHE: A canonical approach for accelerating the evaluation of predicates with UDFs is to directly cache the UDF results at tuple-level (*i.e.*, frame-level) granularity [27, 32]. We implement such a function caching technique in EVA's EXECUTION ENGINE. Specifically, for each UDF, the EXECUTION ENGINE maintains an in-memory hash table that maps the input arguments to the outcomes. It uses xxHash [11] to efficiently compute 128-bit hash values of the input arguments of the UDF.

HARDWARE SETUP. We perform experiments on a server with these specifications: 28 Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz, 1 NVIDIA Quadro P6000 GPU, and 256 GB RAM.

5.2 End-to-End Comparison

HIT PERCENTAGE. Measures the fraction of UDF invocations satisfied using previously materialized results.

$$\text{Hit Percentage} = \frac{\text{number of reused UDF invocations}}{\text{total number of UDF invocations}} * 100$$

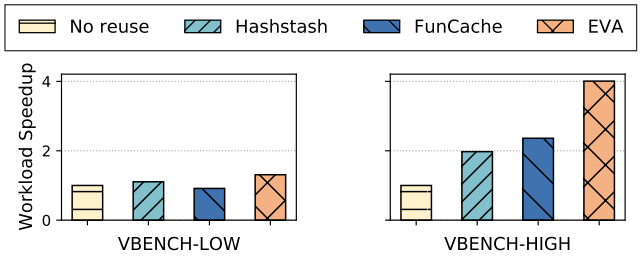


Figure 5: Workload speedup – Impact of reuse algorithms on VBENCH-LOW and VBENCH-HIGH workloads over the MEDIUM-UA-DETRAC video set.

Table 3: UDF Statistics – C_u is the cost of each UDF invocation in VBENCH-HIGH measured in milliseconds per tuple (MEDIUM-UA-DETRAC dataset). For deep learning models (*e.g.*, FASTERRCNNRESNET50), C_u includes the pre- and post-processing time, besides the inference time. We configure the GPU batch size to 20. #DI and #TI represent the number of distinct invocations and the total number of invocations, respectively.

UDF	C_u	#DI	#TI	GPU/CPU
FASTERRCNNRESNET50	99	13,820	72,457	GPU
CARTYPE	6	114,431	414,119	GPU
COLORDET	5	111,631	219,264	CPU

Table 2 presents the hit percentage with different reuse algorithms under different reuse-potential query sets. A higher hit percentage implies that the algorithm can exploit more reuse opportunities based on the results of previous queries within the workload, thereby leading to a lower query execution time (shown in Fig. 5). The most notable observation is that EVA has at least an 11.7× higher hit percentage than HASHSTASH because the sub-tree matching problem in traditional database systems is not geared towards reusing results associated with UDF invocations while evaluating the predicates in the queries. For example, HASHSTASH can only reuse the FASTERRCNNRESNET50 between queries in Table 1, while EVA can further reuse the outcomes of CARTYPE and COLORDET. EVA achieves the same hit percentage as FUNCACHE, which is optimal under both workloads.

WORKLOAD SPEEDUP. Figure 5 presents the workload speed up across different reuse-potential query sets. With NO-REUSE, VBENCH-LOW and VBENCH-HIGH take 0.96 hours and 3.1 hours, respectively. Consistent with the hit percentage, EVA's reuse algorithm lowers execution time by 1.2× over HASHSTASH on VBENCH-LOW and by 2× on VBENCH-HIGH. Though FUNCACHE has the same hit percentage as EVA, EVA outperforms FUNCACHE by 1.7× on VBENCH-HIGH, and FUNCACHE achieves a negative speedup (*i.e.*, 0.95×) on VBENCH-LOW. This is due to the cumulative overhead of hashing the input arguments during every UDF invocation (even with the fast xxHash function). Another limitation of the FUNCACHE is that it is applied during execution time (*i.e.*, when evaluating an UDF). So it does not support optimizations like materialization-aware predicate reordering.

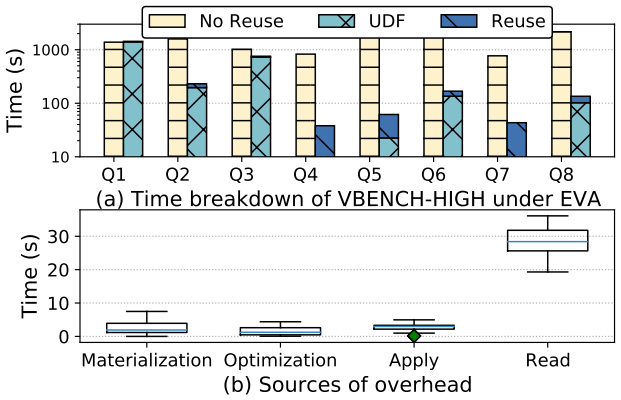


Figure 6: Time Breakdown and Overhead Analysis – (a) showcases the time breakdown (log-scale) of eight queries in VBENCH-HIGH under EVA. (b) shows the box plot of the time spent on materialization, optimization, apply operator, and reading (*i.e.*, video frames and materialized UDF results) for each query. Outliers are marked as green diamonds.

For a given workload, the upper bound on the maximum speedup possible with respect to NO-REUSE is as follows:

$$\text{Workload Speedup} = \frac{\sum_{u \in \text{all UDF invocations}} C_u}{\sum_{u \in \text{all distinct UDF invocations}} C_u + \text{reuse cost}} \tag{7}$$

$$< \frac{\sum_{u \in \text{all UDF invocations}} C_u}{\sum_{u \in \text{all distinct UDF invocations}} C_u}$$

Here, C_u is the cost of invoking u. We compute this upper bound by examining the UDF invocations across all the queries in the workload. Table 3 lists the statistics of UDF invocations under VBENCH-HIGH and MEDIUM-UA-DETRAC. For VBENCH-HIGH, the workload speedup is bounded by 4.11× over the NO-REUSE setting. For VBENCH-LOW, the upper bound is 1.42×. In both cases, EVA delivers a near-optimal speedup (0.97× of the upper bound on VBENCH-HIGH and 0.92× on VBENCH-LOW).

STORAGE FOOTPRINT. The storage footprint associated with materializing the results of UDF invocations for VBENCH-LOW is 12.5 MiB and 14.3 MiB for VBENCH-HIGH. The size of the MEDIUM-UA-DETRAC video dataset is 16 GiB. EVA's reuse algorithm takes up to 0.09 % extra storage space. This is because the UDFs used in the benchmark extract lightweight structured meta-data (*e.g.*, bounding boxes, color, and vehicle type) from the video. So the storage footprint is significantly lower than that of the video itself. This might not be the case for certain UDFs (*e.g.*, video colorization).

5.3 Time Breakdown

To better understand the benefits and overhead of EVA's reuse algorithm, we show the time breakdown for individual queries in Fig. 6. EVA starts from a state with no materialized views, so the first three queries in VBENCH-HIGH incur high UDF execution costs, as shown in Fig. 6 (a), while later queries are much faster. After executing these queries, EVA additionally pays the compute cost of materializing the UDF invocations. Among them, only Q_1 incurs a 0.95× slowdown with respect to NO-REUSE (a tolerable overhead

Table 4: Time Breakdown of Q_6 in VBENCH-HIGH under NO-REUSE and EVA— This table breaks down query processing time into: (1) latency of UDF evaluation, (2) reading the video from the disk, (3) reading the materialized results, (4) materializing the new UDF evaluation results, and (5) other operations (*e.g.*, optimizer, join, crop, *e.t.c.*)

Latency (s)	UDF	Read Video	Read View	Mat	Other
NO-REUSE	997	22	0	0	2
EVA	5	19	10	2	5

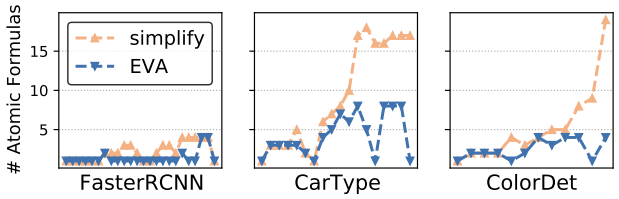


Figure 7: Effectiveness of EVA's Symbolic Predicate Reduction – The x-axis represents the intersection, difference, and union predicates calculated in the OPTIMIZER when executing the VBENCH-HIGH. The y-axis contains the number of atomic formulae in those predicates.

for accelerating subsequent queries). Fig. 6 (a) also demonstrates that the EVA's reuse cost is much lower than the UDF execution cost.

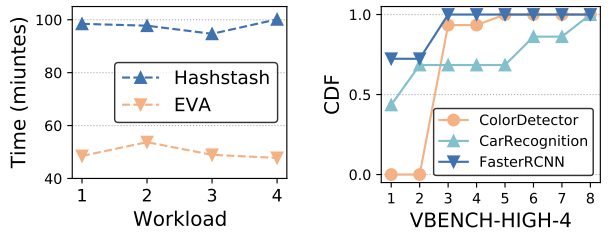
Table 4 presents a fine-grained time breakdown of an exemplar query Q_6. Compared to the NO-REUSE setting, EVA replaces the 997 seconds of UDF evaluation with 10 seconds of reading the materialized results and 5 seconds of UDF evaluation on the remaining input rows. Meanwhile, time spent on materializing new results, query optimization, and join operations is much lower compared to the benefits of EVA's reuse.

OVERHEAD ANALYSIS. Fig. 6 (b) lists the key sources of overhead: (1) materializing the outcomes of UDF invocations; (2) OPTIMIZER analyzing and rewriting the query; (3) adding the APPLY operator for reusing the results of a materialized view; (4) loading frames and materialized results from the storage engine; We do not present certain system components with negligible overhead (*e.g.*, parser). The most notable observation is that the OPTIMIZER has low overhead. This shows that the semantic reuse algorithm and symbolic analysis are efficient. Materializing the outcomes of UDF invocations has a low overhead due to batch-level processing in EVA (batch size = 200 MiB) [4]. In contrast, the time spent on reading tables and views is significant because the conditional apply operator needs to read the complete table to find out missing entries.

5.4 Semantic Reuse Algorithms

SYMBOLIC PREDICATE REDUCTION. In this experiment, we compare EVA's predicate reduction algorithm (§ 4.1) with Sympy's off-the-shelf `simplify` function. The `simplify` function is based on the pattern matching and Quine-McCluskey algorithm [52]. Fig. 7 shows that EVA's algorithm outperforms the `simplify` for all three UDFs' predicate analyses. This is because the pattern matching logic in SymPy's `simplify` cannot extensively support

[4]CPU batch size is different from the GPU batch size in Table 3.



(a) Execution Time (b) Materialized UDF Results

Figure 8: Impact of Order of Queries – (a) presents the execution time of four random permutations of VBENCH-HIGH with HASHSTASH and EVA's reuse algorithms. (b) presents how the materialized UDF results in EVA converge over queries in the fourth permutation.

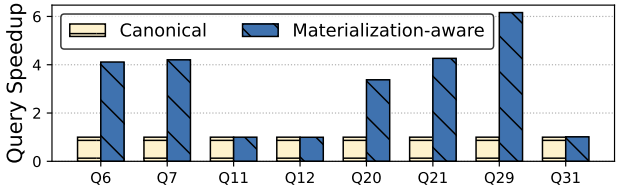


Figure 9: Impact of Materialization-Aware Predicate Reordering – Query speedup with canonical and materialization-aware ranking functions.

the interactions between inequalities and logic operations. In contrast, EVA leverages Sympy's inequality solver in the predicate reduction procedure.

In Fig. 7, the gap between the `simplify` and EVA's algorithm is smaller for FASTERRCNNRESNET50 than CARTYPE or COLORDET. This is because in VBENCH-HIGH, the associated predicates for FASTERRCNNRESNET50 only involve the id column. In contrast, the associated predicates for CARTYPE or COLORDET contain up to four variables (*i.e.*, id, label, area, and the other UDF). `simplify`'s reduction does not work well with polyadic predicates. Further, when the `simplify` function fails to reduce a predicate, it cannot recover in subsequent queries, and the predicate becomes more complicated over time. This explains why in Fig. 7, `simplify` leads to an extraordinarily high number of atomic formulae for VEHICLEMODEL and VEHICLECOLOR.

MATERIALIZATION-AWARE PREDICATE REORDERING. To show the effectiveness of materialization-aware predicate reordering, we construct four workloads (VBENCH-HIGH- 1, 2, 3 and 4) that are random permutations of queries in VBENCH-HIGH. The rationale is that the amount of materialized results that individual UDF invocation can reuse differs in every permutation. So, the cost of UDF should not be static. Fig. 8 (a) shows that EVA's reuse algorithms lowers execution time by at least 1.8× over HASHSTASH (stronger baseline). In workloads 1, 3, and 4, where the predicate reordering is beneficial, EVA outperforms HASHSTASH by 2×. To discuss why predicate reordering is not useful in the second workload, we compare the execution time of the queries with two predicate reordering algorithms: (1) using a canonical ranking function (Eq. (2)), and (2) using a materialization-aware ranking function (Eq. (4)). The results are shown in Fig. 9. We only list the queries with multiple

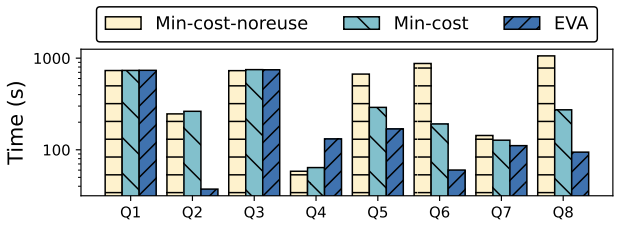


Figure 10: Impact of Logical UDF Reuse – Comparison of execution time (log scale) against baselines that directly substitute the logical UDF with the least expensive physical UDF that satisfies the accuracy constraints.

predicates across all permutations. Table 1 shows an example query (Q_6) with two UDF predicates, namely ColorDet and CarType. Materialization-aware predicate reordering accelerates queries by 3–6 × on most queries. This is because the canonical ranking function only considers the execution cost of the UDFs, whereas the materialization-aware ranking function also factors in the available materialized results. With queries Q_{11}, Q_{12}, and Q_{31}, both ranking functions return the same predicate ordering. We attribute this to the fact that the UDFs that result in lower ranks using the canonical ranking function also have a higher fraction of required results materialized.

Impact of Logical UDF Reuse. In this experiment, we evaluate the benefits of logical UDF reuse optimization. In earlier experiments, for a fair comparison with other baselines, all the queries in the vbench referred to an actual physical model (faster-rcnn). In this experiment, we replace all the occurrences of faster-rcnn in the workload with a logical UDF (*i.e.*, ObjectDetector). We consider three physical UDFs, namely, (1) yolo-tiny [2], (2) FasterRCNNResnet50, and (3) FasterRCNNResnet101 [54]. Table 5 lists these model's accuracy and inference cost on the COCO [39] dataset. We pick these models as they are readily available in popular object detection libraries [34, 61]. The workload emulates multiple interactive video analytics applications with different accuracy requirements. We compare Algorithm 2) against two baselines: min-cost-noreuse (reuse disabled) and min-cost (reuse enabled). In both baselines, we substitute the logical UDF with the least expensive physical UDF that satisfies the accuracy constraint. For example, if the required accuracy is low, we substitute it with yolo-tiny. Fig. 10 shows the results. With Q_2 (low accuracy requirement), EVA is 6.6× faster than both baselines because EVA reuses results of FasterRCNNResnet50 from Q_1, whereas min-cost substitutes with the least expensive model (yolo-tiny) and thus has no reuse opportunity. With queries $Q_6 - Q_8$, EVA achieves a speed up in the range of 1.2–3.2× compared to min-cost. This is because EVA reuses results from multiple views, whereas min-cost only reuses results from the minimum cost UDF. For example, yolo-tiny may reuse results from FasterRCNNResnet50 and FasterRCNNResnet101. EVA considers all such reuse opportunities. With Q_4, EVA is 2× slower than both baselines. This is because EVA reuses results from a high accuracy model, which results in more objects being detected. Thus, subsequent dependent UDF (*e.g.*, VehicleModel) must be evaluated for more objects, thus increasing the overall query time. We discuss this limitation in § 6.

Table 5: Statistics of the UDF used in logical reuse experiment. We use a batch size of 20. C_u is the cost of each UDF in milliseconds per tuple. Accuracy values are boxAP on COCO.

	C_u(ms)	Accuracy
yolo-tiny	9	17.6 (LOW)
FasterRCNNResnet50	99	37.9 (MEDIUM)
FasterRCNNResnet101	120	42.0 (HIGH)

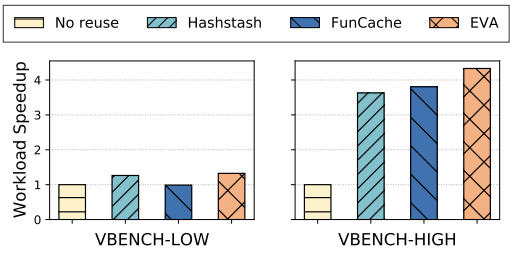


Figure 11: Impact of Video Content – It shows the workload speedup on the jackson video dataset.

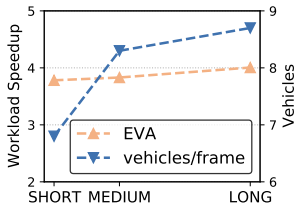


Figure 12: Impact of Video Length – The left y-axis shows the workload speedup and the right y-axis shows the average number of vehicles per frame across differently-sized ua-detrac videos.

5.5 Impact of Video Content and Length

We next examine how the content of the video affects the performance gains of EVA. Fig. 11 presents the workload speedup of EVA and other baselines on the jackson video dataset. With No-Reuse, vbench-low and vbench-high take 0.53 and 1.7 hours, respectively. While EVA still outperforms both baselines, the gap is smaller because this dataset has significantly fewer vehicle objects (0.1 vehicles per frame), leading to fewer ColorDet and CarType invocations (reused by EVA). Thus, the benefits of EVA are more prominent on workloads that contain more frequent UDF invocations in the predicates.

Video Length. To study how the benefits of EVA vary with the length of the video, we measure the workload speedup of vbench-high query set on SHORT- and LONG-ua-detrac. We alter the query set to scale the *id* predicate range to the length of these videos. For instance, $id < 10000$ on medium-ua-detrac translates to $id < 5000$ for short-ua-detrac and $id < 20000$ for long-ua-detrac, respectively. Fig. 12 shows that the workload speedup does not drop with longer videos, demonstrating the scalability of EVA. This is because the vehicle objects are nearly uniformly distributed across frames in the ua-detrac videos. So, the video length does

not significantly impact the speedup (Eq. (7)). The slight increase in workload speedup in Fig. 12 stems from higher number of average vehicles per frame in LONG-UA-DETRAC.

5.6 Impact of Specialized Filters

We next examine how the reuse algorithm works in conjunction with specialized filters [35, 40]. These specialized filters return a boolean decision that decides whether the frame needs to be subsequently processed by an expensive UDF. In this experiment, we use a lightweight DNN model with two convolutional layers as a specialized filter. Since these filters are lightweight UDFs, we also materialize their results whenever possible.

We consider two configurations: (1) EVA: reuse enabled but no specialized filters, and (2) EVA+Filter: reuse enabled with specialized filter. The experiment is performed on JACKSON video because the filtering works best on videos with a low percentage of average vehicles per frame [35]. Execution time with EVA and EVA+Filter configurations are 1393 s and 1075 s, respectively (1.3× speedup). This reduction in execution time is in addition to the 4× speedup EVA delivers without using filters (Fig. 11). We attribute this additional gain to reducing the invocation of expensive UDF by filtering out irrelevant frames using the lightweight UDF. This experiment illustrates that reuse is orthogonal to the filtering optimization used in other recently proposed VDBMSs [35, 40].

6 LIMITATIONS

We now discuss the limitations of EVA and present our ideas that may address the problems in the future work.

Symbolic analysis of join predicates. Join predicates increase the complexity of identifying UDF-centric reuse opportunities. Consider the following query plans:

- $Q_1 : \Pi_{UDF(A.col,B.col)}(A \bowtie_{A.id=B.id} B)$
- $Q_2 : \Pi_{UDF(A.col,B.col)}(A \bowtie_{A.id=B.id+1} B)$
- $Q_3 : \Pi_{UDF(A.col,B.col)}(A \bowtie_{A.id \equiv B.id \mod 2} B)$

where table A and B are heterogeneous except for the *id* column. Here, no reuse opportunities exist between Q_1 and Q_2, while Q_1 subsumes Q_3. While it is possible to do symbolic analysis of join predicates, EVA currently does not support it.

Chained function calls and Fuzzy matching. In case of logical UDF reuse, the selection of a physical UDF may affect the execution cost of a subsequent UDF. For example, as FASTER-RCNN detects more objects than YOLO-TINY, substituting the object detector with FASTER-RCNN produces more objects, thereby increasing the number of evaluations of dependent UDFs (*e.g.*, VEHICLECOLOR, VEHICLEMODEL). Taking this into consideration in the cost model will further improve performance. Another observation is that the bounding boxes detected by different object detection models for the same object are likely to be spatially close to each other. We plan to extend EVA to fuzzily reuse the results of VEHICLEMODEL UDF on similar bounding boxes in the future.

7 RELATED WORK

Visual DBMSs. Researchers have presented techniques for efficiently analyzing visual data for several decades. These include a rank-join operator for multi-feature image similarity matching [5, 28], support of streaming media [4, 23]. More recently, BlazeIt [35] utilizes specialized neural networks to accelerate aggregation and limit queries. It supports a declarative query language for analyzing spatio-temporal features of the video. NoScope [36] reduces execution cost by leveraging cheap filters. [40] uses probabilistic predicates to accelerate machine learning inference. Tahoma [3] relies on classifier cascades to speed up visual analytics queries. Weld [47] introduces a common runtime that optimizes data operations among existing analytics libraries. All these systems are orthogonal to EVA and can be coupled to accelerate video analytical queries further.

Reuse Algorithms in Traditional DBMSs. Researchers have extensively studied algorithms for reusing results in traditional DBMSs [13, 17, 19, 27, 31, 33, 45, 50, 55, 56, 64]. Similar techniques have been proposed in non-relational data processing systems [1, 14, 22, 53]. The key limitation of most of these algorithms is that they take a syntax-based approach to select sub-expressions to materialize and map it to a cost-based optimization problem. Since EVA takes a semantics-based approach to reusing results, it is more effective in leveraging opportunities for reusing results(*e.g.*, non-exact reuse in simple predicates that do not contain UDFs, and compound predicates with logic and arithmetic expressions).

Traditional DBMSs [46] use a set of complex query rewrite rules for leveraging materialized views. These rewrite rules are complementary to the ones tailored for UDFs in EVA.

Symbolic Computation in DBMSs. Researchers have proposed several applications of symbolic computation in DBMSs. These include: (1) systems for verifying or disproving the equivalence of SQL queries [9, 65], (2) systems to automatically generate input tables and parameter values for database applications [57]. EVA leverages symbolic computation to analyze UDF-based predicates.

Storage Systems for Video Analytics. VSS [25], VStore [62], and TASM [12] are novel storage engines tailored for video analytics. Since EVA supports a pluggable storage engine architecture, leveraging these specialized storage engines can further reduce query execution time.

8 CONCLUSION

We presented EVA, a VDBMS for accelerating exploratory video analytics using materialized views. EVA adopts a novel symbolic approach to analyze the degree of reuse across queries and applies a series of rule-based transformations geared towards reusing UDF results. It leverages a materialization-aware ranking function for reordering predicates and employs a logical UDF reuse optimization tailored for video analytics. Our empirical analysis of EVA shows that it outperforms the SOTA reuse algorithms on exploratory video analytics workloads by 4× with a negligible storage overhead.

ACKNOWLEDGMENTS

This work was supported in part by the U.S. National Science Foundation (IIS-1850342, IIS-1908984, CNS-1909346, CNS-2008368), Alibaba Innovative Research (AIR) Program, Cisco, Adobe, Intel, and a gift from Microsoft Corp. We thank colleagues in Georgia Tech Database Group and Embedded Pervasive Lab for their constructive feedback in improving the system.

REFERENCES

[1] D. Abadi, Yanif Ahmad, M. Balazinska, U. Çetintemel, Mitch Cherniack, J. Hwang, W. Lindner, Anurag Maskey, A. Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and S. Zdonik. 2005. The Design of the Borealis Stream Processing Engine. In *CIDR*.

[2] P. Adarsh, Pratibha Rathi, and M. Kumar. 2020. YOLO v3-Tiny: Object Detection and Recognition using one stage improved model. *2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS)* (2020), 687–694.

[3] Michael R. Anderson, Michael J. Cafarella, G. Ros, and T. Wenisch. 2019. Physical Representation-Based Predicate Optimization for a Visual Analytics Database. *2019 IEEE 35th International Conference on Data Engineering (ICDE)* (2019), 1466–1477.

[4] W. Aref, A. Catlin, A. Elmagarmid, Jianping Fan, J. Guo, M. Hammad, I. Ilyas, M. Marzouk, S. Prabhakar, A. Rezgui, S. Teoh, E. Terzi, Yi-Cheng Tu, A. Vakali, and Xingquan Zhu. 2002. A distributed database server for continuous media. *Proceedings 18th International Conference on Data Engineering* (2002), 490–491.

[5] W. Aref, A. Catlin, Jianping Fan, A. Elmagarmid, M. Hammad, I. Ilyas, M. Marzouk, and Xingquan Zhu. 2002. A Video Database Management System for Advancing Video Database Research. In *Multimedia Information Systems*.

[6] Michael Bayer. 2012. SQLAlchemy. In *The Architecture of Open Source Applications Volume II: Structure, Scale, and a Few More Fearless Hacks*, Amy Brown and Greg Wilson (Eds.). aosabook.org. http://aosabook.org/en/sqlalchemy.html

[7] David Buchfuhrer and Christopher Umans. 2008. The complexity of boolean formula minimization. In *International Colloquium on Automata, Languages, and Programming*. Springer, 24–35.

[8] S. Chaudhuri and Kyuseok Shim. 1999. Optimization of queries with user-defined predicates. In *TODS*.

[9] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL.. In *CIDR*.

[10] V. Chvatal. 1979. A Greedy Heuristic for the Set-Covering Problem. *Math. Oper. Res.* 4, 3 (Aug. 1979), 233–235. https://doi.org/10.1287/moor.4.3.233

[11] Yann Collet. 2021. xxHash - Extremely fast hash algorithm. Retrieved Apr 17, 2021 from https://github.com/Cyan4973/xxHash

[12] Maureen Daum, Brandon Haynes, Dong He, Amrita Mazumdar, M. Balazinska, and Alvin Cheung. 2020. TASM: A Tile-Based Storage Manager for Video Analytics. *ArXiv* abs/2006.02958 (2020).

[13] K. Dursun, Carsten Binnig, U. Çetintemel, and Tim Kraska. 2017. Revisiting Reuse in Main Memory Database Systems. *Proceedings of the 2017 ACM International Conference on Management of Data* (2017).

[14] Iman Elghandour and Ashraf Aboulnaga. 2012. ReStore: Reusing Results of MapReduce Jobs. *Proc. VLDB Endow.* 5 (2012), 586–597.

[15] Mostafa Elhemali, C. Galindo-Legaria, T. Grabs, and Milind Joshi. 2007. Execution strategies for SQL subqueries. In *SIGMOD '07*.

[16] Uriel Feige. 1998. A Threshold of Ln n for Approximating Set Cover. *J. ACM* 45, 4 (July 1998), 634–652. https://doi.org/10.1145/285055.285059

[17] Alex Galakatos, Andrew Crotty, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. 2017. Revisiting Reuse for Approximate Query Processing. *Proc. VLDB Endow.* 10 (2017), 1142–1153.

[18] C. Galindo-Legaria and Milind Joshi. 2001. Orthogonal optimization of subqueries and aggregation. In *SIGMOD '01*.

[19] G. Giannikis, Darko Makreshanski, G. Alonso, and D. Kossmann. 2014. Shared Workload Optimization. *Proc. VLDB Endow.* 7 (2014), 429–440.

[20] G. Graefe. 1995. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.* 18 (1995), 19–29.

[21] R. Gruener, O. Cheng, and Y. Litvin. 2018. Introducing Petastorm: Uber ATG's Data Access Library for Deep Learning. Retrieved May 7, 2021 from https://eng.uber.com/petastorm/

[22] P. Gunda, L. Ravindranath, C. Thekkath, Y. Yu, and Li Zhuang. 2010. Nectar: Automatic Management of Data and Computation in Datacenters. In *OSDI*.

[23] M. Hammad, W. Aref, and A. Elmagarmid. 2002. Search-based buffer management policies for streaming in continuous media servers. *Proceedings. IEEE International Conference on Multimedia and Expo* 1 (2002), 253–256 vol.1.

[24] Michael Z. Hanani. 1977. An optimal evaluation of Boolean expressions in an online query system. *Commun. ACM* 20 (1977), 344–347.

[25] Brandon Haynes. 2021. VSS: A Storage System for Video Analytics.

[26] J. Hellerstein. 1994. Practical predicate placement. In *SIGMOD '94*.

[27] J. Hellerstein and M. Stonebraker. 1993. Predicate migration: optimizing queries with expensive predicates. In *SIGMOD '93*.

[28] I. Ilyas, W. Aref, and A. Elmagarmid. 2002. Joining Ranked Inputs in Practice. In *VLDB*.

[29] Wolfram Research, Inc. [n.d.]. Mathematica, Version 12.3.1. https://www.wolfram.com/mathematica Champaign, IL, 2021.

[30] Y. Ioannidis. 2003. The History of Histograms (abridged). In *VLDB*.

[31] M. Ivanova, M. Kersten, N. Nes, and R. Goncalves. 2009. An architecture for recycling intermediates in a column-store. *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data* (2009).

[32] A. Jhingran. 1988. A Performance Study of Query Optimization Algorithms on a Database System Supporting Procedures. In *VLDB*.

[33] Alekh Jindal, Konstantinos Karanasos, S. Rao, and Hiren Patel. 2018. Selecting Subexpressions to Materialize at Datacenter Scale. *Proc. VLDB Endow.* 11 (2018), 800–812.

[34] Glenn Jocher, Yonghye Kwon, Guigarfr, Perry0418, Josh Veitch-Michaelis, Ttayu, Daniel Suess, Fatih Baltacı, Gabriel Bianconi, IlyaOvodov, , Marc, E96031413, Chang Lee, Dustin Kendall, , Falak, Francisco Reveriano, , FuLin, GoogleWiki, Jason Nataprawira, Jeremy Hu, LinCoce, LukeAI, NanoCode012, NirZarrabi, Oulbacha Reda, Piotr Skalski, SergioSanchezMontesUAM, Shiwei Song, Thomas Havlik, and Timothy M. Shead. 2021. ultralytics/yolov3: v9.5.0 - YOLOv5 v5.0 release compatibility update for YOLOv3. https://doi.org/10.5281/ZENODO.4681234

[35] Daniel Kang, Peter Bailis, and M. Zaharia. 2019. BlazeIt: Optimizing Declarative Aggregation and Limit Queries for Neural Network-Based Video Analytics. *Proc. VLDB Endow.* 13 (2019), 533–546.

[36] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and M. Zaharia. 2017. NoScope: Optimizing Neural Network Queries over Video at Scale. *arXiv: Databases* (2017).

[37] Daniel Kang, John Guibas, Peter Bailis, Tatsunori Hashimoto, Yi Sun, and Matei Zaharia. 2021. Accelerating Approximate Aggregation Queries with Expensive Predicates. *Proc. VLDB Endow.* 14, 11 (jul 2021), 2341–2354. https://doi.org/10.14778/3476249.3476285

[38] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. 1989. Architecture and performance of relational algebra machine GRACE.

[39] Tsung-Yi Lin, M. Maire, Serge J. Belongie, James Hays, P. Perona, D. Ramanan, Piotr Dollár, and C. L. Zitnick. 2014. Microsoft COCO: Common Objects in Context. In *ECCV*.

[40] Y. Lu, Aakanksha Chowdhery, Srikanth Kandula, and S. Chaudhuri. 2018. Accelerating Machine Learning Inference with Probabilistic Predicates. *Proceedings of the 2018 International Conference on Management of Data* (2018).

[41] Carsten Lund and Mihalis Yannakakis. 1994. On the Hardness of Approximating Minimization Problems. *J. ACM* 41, 5 (Sept. 1994), 960–981. https://doi.org/10.1145/185675.306789

[42] Wes McKinney. 2010. Data Structures for Statistical Computing in Python.

[43] Aaron Meurer, C. Smith, Mateusz Paprocki, O. Certík, S. B. Kirpichev, M. Rocklin, A. Kumar, Sergiu Ivanov, J. K. Moore, Sartaj Singh, T. Rathnayake, Sean Vig, B. Granger, R. Muller, F. Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, M. Curry, A. Terrel, S. Roucka, A. Saboo, Isuru Fernando, Sumith Kulal, R. Cimrman, and A. Scopatz. 2017. SymPy: symbolic computing in Python. *PeerJ Comput. Sci.* 3 (2017), e103.

[44] Oscar Moll, F. Bastani, Sam Madden, M. Stonebraker, V. Gadepally, and Tim Kraska. 2020. ExSample: Efficient Searches on Video Repositories through Adaptive Sampling. *ArXiv* abs/2005.09141 (2020).

[45] F. Nagel, P. Boncz, and Stratis Viglas. 2013. Recycling in pipelined query evaluation. *2013 IEEE 29th International Conference on Data Engineering (ICDE)* (2013), 338–349.

[46] Oracle. 2017. Advanced Query Rewrite for Materialized Views. Retrieved Dec 11, 2020 from https://docs.oracle.com/database/121/DWHSG/qradv.htm#DWHSG08026

[47] Shoumik Palkar, J. Thomas, D. Narayanan, Pratiksha Thaker, R. Palamuttam, Parimarjan Negi, A. Shanbhag, Malte Schwarzkopf, H. Pirk, Saman P. Amarasinghe, S. Madden, and M. Zaharia. 2018. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. *Proc. VLDB Endow.* 11 (2018), 1002–1015.

[48] T. Parr, Sam Harwell, and Kathleen Fisher. 2014. Adaptive LL(*) parsing: the power of dynamic analysis. In *OOPSLA*.

[49] Adam Paszke, S. Gross, Francisco Massa, A. Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Z. Lin, N. Gimelshein, L. Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*.

[50] L. Perez and C. Jermaine. 2014. History-aware query optimization with materialized intermediate views. *2014 IEEE 30th International Conference on Data Engineering* (2014), 520–531.

[51] Viswanath Poosala, Yannis E. Ioannidis, Peter J. Haas, and Eugene J. Shekita. 1996. Improved histograms for selectivity estimation of range predicates. In *SIGMOD '96*.

[52] Willard V Quine. 1952. The problem of simplifying truth functions. *The American mathematical monthly* 59, 8 (1952), 521–531.

[53] Lana Ramjit, Matteo Interlandi, Eugene Wu, and Ravi Netravali. 2019. Acorn: Aggressive Result Caching in Distributed Data Processing Frameworks. *Proceedings of the ACM Symposium on Cloud Computing* (2019).

[54] Shaoqing Ren, Kaiming He, Ross B. Girshick, and J. Sun. 2015. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39 (2015), 1137–1149.

[55] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. 2000. Efficient and extensible algorithms for multi query optimization. *ArXiv* cs.DB/9910021 (2000).

[56] K. Tan, S. Goh, and B. Ooi. 2001. Cache-on-demand: recycling with certainty. *Proceedings 17th International Conference on Data Engineering* (2001), 633–640.

[57] Margus Veanes, Nikolai Tillmann, and Jonathan De Halleux. 2010. Qex: Symbolic SQL query explorer. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 425–446.

[58] Abdul Wasay, Xinding Wei, Niv Dayan, and Stratos Idreos. 2017. Data Canopy: Accelerating Exploratory Statistical Analysis. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 557–572. https://doi.org/10.1145/3035918.3064051

[59] Longyin Wen, Dawei Du, Zhaowei Cai, Z. Lei, Ming-Ching Chang, H. Qi, Jongwoo Lim, Ming-Hsuan Yang, and Siwei Lyu. 2020. UA-DETRAC: A new benchmark and protocol for multi-object detection and tracking. *Comput. Vis. Image Underst.* 193 (2020), 102907.

[60] Wikipedia contributors. 2021. Computer algebra — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Computer_algebra&oldid=1000609796. [Online; accessed 23-August-2021].

[61] Yuxin Wu, Alexander Kirillov, Francisco Massa, Wan-Yen Lo, and Ross Girshick. 2019. Detectron2. https://github.com/facebookresearch/detectron2.

[62] Tiantu Xu, Luis Materon Botelho, and F. Lin. 2019. VStore: A Data Store for Analytics on Large Videos. *Proceedings of the Fourteenth EuroSys Conference 2019* (2019).

[63] Y. Xu. 1998. Efficiency In The Columbia Database Query Optimizer.

[64] Jingren Zhou, P. Larson, J. Freytag, and Wolfgang Lehner. 2007. Efficient exploitation of similar subexpressions for query processing. In *SIGMOD '07*.

[65] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Dong Xu. 2019. Automated verification of query equivalence using satisfiability modulo theories. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1276–1288.