Optimizing Video Queries with Declarative Clues

Daren Chao
University of Toronto
Toronto, Canada
drchao@cs.toronto.edu

Yueting Chen
York University
Toronto, Canada
chenyt@yorku.ca

Nick Koudas
University of Toronto
Toronto, Canada
koudas@cs.toronto.edu

Xiaohui Yu
York University
Toronto, Canada
xhyu@yorku.ca

ABSTRACT

Video Database Management Systems (VDBMS) leverage advancements in computer vision and deep learning for efficient video data analysis and retrieval. This paper introduces the concept of user-specified Clues, allowing users to incorporate domain-specific knowledge, referred to as Clues, into query optimization. Clues are expressed as Clue types, each associated with optimization rules, and applied to queries through Clue instances. The extensible ClueVQS system we present to incorporate these ideas, optimizes queries automatically, utilizing Clues to improve processing efficiency. We also introduce algorithms to optimize queries using Clues allowing for trade-offs between speed and query accuracy. Our proposals and system address challenges such as data-dependent Clue effectiveness, limiting search space, and accuracy-efficiency trade-offs. Detailed experimental results demonstrate query speedups of up to two orders of magnitude compared to other applicable approaches, and a reduction of the query optimizer time by up to 95% while respecting user-specified accuracy constraints, showcasing the effectiveness of the proposed framework.

PVLDB Reference Format:
Daren Chao, Yueting Chen, Nick Koudas, and Xiaohui Yu. Optimizing Video Queries with Declarative Clues. PVLDB, 17(11): 3256-3268, 2024.
doi:10.14778/3681954.3681998

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/daren996/ClueVQS.

1 INTRODUCTION

Video Database Management Systems (VDBMS), fueled by advancements in computer vision and deep learning, have actively been developed to facilitate the analysis and retrieval of video data [20, 31, 39, 40, 43]. These systems enable the search and analysis of video content using SQL-like queries incorporating various predicates. Recently several approaches have been presented to efficiently analyze large-scale visual data in VDBMS, including sampling-based approaches, filtering, as well as reuse of intermediate outcomes [4, 8, 11, 16, 21, 25, 28, 30, 34, 41, 49, 50].

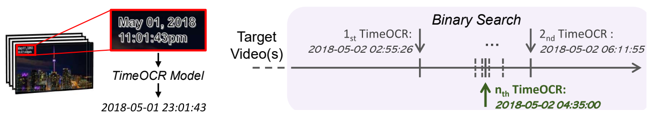
Videos vary greatly in content, and more often than not, users have an accurate idea of the content or structure of the video they wish to analyze. Such knowledge could be utilized to greatly facilitate query processing but is currently overlooked. To realize things concretely, consider the following example query Q1.

Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:10.14778/3681954.3681998

```sql
-- Q1: Find a frame from a time-lapse video.
SELECT DISTINCT fid FROM LOAD_VIDEO('time_lapse.mp4')
  WHERE TimeOCR(img) = ToDate('2018-05-02 04:35:00');
```

Listing 1: Example Query Q1.

Q1 aims to locate a specific frame in a time-lapse video that matches a given timestamp, using a model TimeOCR[1] to detect the timestamp on each frame, as depicted in Fig. 1a. Since the user has the knowledge that timestamps appear in each frame, this information can be capitalized to process the query faster. In particular, if we know timestamps increase monotonically over the sequence of the video frames, leveraging this knowledge an *optimization strategy* can be implemented, enabling a frame binary search technique, as illustrated in Fig. 1b. The TimeOCR model is selectively applied to frames at middle points during the binary search process, thereby avoiding invoking the model on a frame-by-frame basis.


May 01, 2018
11:01:43pm
TimeOCR Model
2018-05-01 23:01:43
Target Video(s)
Binary Search
1st TimeOCR: 2018-05-02 02:55:26
...
2nd TimeOCR : 2018-05-02 06:11:55
nth TimeOCR: 2018-05-02 04:35:00

(a) TimeOCR model can recognize the timestamp. **(b) A binary search strategy can be applied to the monotonic timestamps to minimize model calls.**

Figure 1: Optimization of Q1 utilizing the knowledge about the monotonic nature of timestamps.

This is just one example of user knowledge that is currently difficult to incorporate into a query framework for videos. In this paper, we refer to such knowledge that can lead to query optimization strategies as Clues. However, VDBMS cannot infer and apply such Clues automatically. Expressing such knowledge through SQL is not an option[2]. Hard-coding such knowledge per query is also impractical. To address this challenge, we build an extensible framework to enable utilization of Clue. We provide the capability to register a Clue type and develop and implement an optimization *rule*[3] for each Clue type. For example, the Clue type conceptualized from the example in Fig. 1b, namely MONOTONIC, along with its optimization rule, can be defined as presented below.

```sql
CREATE CLUE TYPE MONOTONIC (monotonic_on VARCHAR,
    decreasing BOOLEAN, nullable BOOLEAN DEFAULT TRUE)
AS IMPL 'clues/monotonic.py';
```

Listing 2: Clue Type MONOTONIC – Its arguments (e.g. monotonic_on) are designated when instantiating the Clue type for a specific query; its optimization rule (frame binary search) is implemented in a python file in this example.

Then for a specific query like Q1 (Lst. 1), we can declare a Clue *instance* of type MONOTONIC, as presented below. The Clue instances

[1]Optical Character Recognition (OCR) is a technology to recognize text in an image.
[2]SQL is a declarative language that is limited in its ability to handle complex logic beyond simple database queries.
[3]The optimization rules operate on the query plans created from user queries and generate optimized plans for execution.

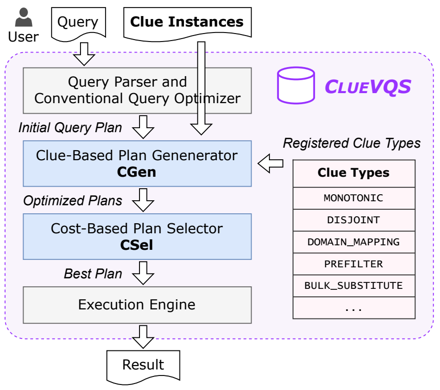


Figure 2: ClueVQS – A user submits a query and a set of Clue instances. CGen applies these Clue instances onto the initial query plan (created by a query parser) using pre-implemented optimization rules, generating multiple optimized plans. CSel selects the best plan with the lowest estimated cost, and its executed results are returned to the user.

are positioned directly following the query statement, indicating the specific query they can be applied to.

```
CREATE CLUE INSTANCE Q1_CLUE1 {
  clue_type: MONOTONIC, monotonic_on: TimeOCR(img),
  decreasing: FALSE, nullable: TRUE};
```

Listing 3: Clue Instance Q1_CLUE1 for Query Q1.

Although Clue instances are query-specific, the rules implemented for the Clue types they belong to are query-independent. This allows each Clue instance with customized arguments to apply the implemented optimization rule for the type it belongs, to a specific query, thus facilitating the utilization of user knowledge in the query.

In this paper, we fully develop the concept of Clue types and embed it into a general system, ClueVQS (Clue-based Video Query Processing System), to support optimizations involving Clues. We demonstrate how Clue types can express existing and new primitives for optimization and we present five Clue types, each equipped with its corresponding optimization rules. Notably, ClueVQS is an *extensible* system that enables easy creation of new Clue types, through the syntax as exemplified in Lst. 2. At query processing time, we submit a set of Clue instances that are potentially beneficial to the query (each belonging to one of the Clue types supported by the system). The system will automatically employ these Clue instances utilizing the corresponding optimization rules to optimize the query.

To fully utilize Clues in query processing, however, we must overcome a number of challenges. First, the effectiveness of a Clue instance to a query is data (video) dependent and difficult for the user to assess. To see this, consider the case where the query involves accurate detection and tracking of a certain type of objects, say, cars. It is possible to apply to the query a Clue instance that utilizes a cheap albeit inaccurate machine learning model (predicate) as a filter to detect whether a frame contains an object type, say, a car. When processing the query, this predicate is applied to every frame first, and a more expensive and accurate predicate is applied if the cheap filter indicates the presence of a car [25, 34, 41]. It is easy to see that the performance of query processing may benefit from, or get penalized by, the introduction of this new Clue instance depending on how many frames contain cars. Arguably, it is difficult, if not impossible, to accurately assess the effectiveness of Clue instances to queries. Instead of relying on the user to decide which Clue instances should be applied, we allow the declaration of Clue instances for a query without worrying about their effectiveness. We design ClueVQS to automatically select the best subset of Clue instances declared by a user for query optimization. Specifically, we propose a plan generator, CGen, that enumerates all query plans (with varying efficiency) examining subsets of Clue instances while optimizing a query. We also propose a plan selector, CSel, that selects the best plan with the lowest cost. This is accomplished by computing the cost of each plan through a selective temporally-constraint execution over the target video.

Second, as the number of Clue instances increases, the search space for candidate query plans grows exponentially. Computing the cost of the candidate plans precisely leads to considerable extra overhead. To address this challenge, our proposed CSel adopts a pruning-based approach. It estimates the cost bounds of candidate query plans through strategically executing them, rather than executing them exhaustively, thereby minimizing time overheads.

Third, since machine learning models incur prediction errors, some Clue instances incorporating such models may reduce query result accuracy. Balancing accuracy and efficiency is crucial to ensure that the improvements in processing efficiency do not detrimentally affect the reliability and accuracy of the query results. To address this challenge, we incorporate an accuracy constraint into our system, representing the maximum acceptable reduction in accuracy resulting from the application of Clues. Furthermore, we refine CSel in our design, proposing CSel-AC, which is tailored to efficiently identify the best plan that conforms to set accuracy constraints.

The entire framework is illustrated in Fig. 2. Our system incorporates a number of Clue types each with its own optimization rules. It is fully extensible, so new types along with associated optimization rules can be incorporated. A query is accompanied with a number of Clue instances (both provided by a user). Queries are optimized automatically to make use of all applicable Clue type optimizations before execution. The system also incorporates user defined accuracy constraints.

In summary, we make the following contributions:

- We propose to incorporate user-provided knowledge into the video query processing framework by introducing the concept of user specified Clue.
- We define Clue types to express this knowledge along with optimization rules to make this knowledge actionable during query optimization.
- We provide the ability to express Clue types applicable to a query by defining and specifying Clue instances.
- We design a system, ClueVQS, that utilizes Clue instances for user queries effectively and automatically to improve the efficiency of video query processing.
- We elaborate on five Clue types along with their specific optimization rules implemented in ClueVQS, while also emphasizing the extensibility by adding new Clue types into the system along with corresponding optimization rules in the future.
- We frame the challenges of identifying the best query plan from a search space of candidate plans as an optimization

problem, and propose a novel efficient optimizer, including a Clue-based plan generator, CGen, and a plan selection algorithm, CSel, as well as its enhanced variant, CSel-AC, that integrates accuracy constraints into the optimization system.
- We present the results of a comprehensive evaluation demonstrating that our framework, by utilizing Clues, can accelerate query runtime by up to 2 orders of magnitude compared to other options. Additionally, it can reduce the optimization time by up to 95%, compared to alternatives, with or without the presence of accuracy constraints.

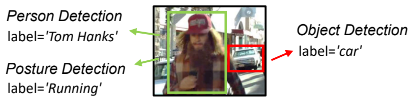
2 BACKGROUND

Video Database Management Systems (VDBMS) utilize User-Defined Functions (UDFs) to leverage advanced machine learning (ML) or computer vision (CV) models for video content retrieval [49]. The UDFs, however, significantly increase the costs associated with query processing in VDBMS. Traditional optimization techniques in relational systems have limited applicability in a video setting, since most predicates involve expensive UDFs [50]. In addition, in contrast to traditional RDBMS whose indexes are created on some columns and automatically maintained, in VDBMS, the input video is naturally ordered by frames, and no other indexes are available prior to invoking external UDFs.

Recent research has been exploring the various approaches to optimizing query processing in VDBMS [25, 34, 41, 49, 50]. Prior to presenting Clues, this section will first provide an overview of UDFs, the execution pipeline, and prior research on query processing and optimization in VDBMS. To illustrate these concepts, we will use the query Q2 that retrieves frames satisfying some specific predicates from a movie, as shown in Lst. 4, as an exemplary case.

Scalar and Table UDFs in VDBMS. In RDBMS (e.g., IBM [22], Snowflake [44], and Databricks [18]), UDFs can be categorized into *scalar* functions (returning a single value) and *table* functions (returning data in a tabular format). These can be similarly adopted in the context of VDBMS. The table functions take a table (such as OBJ_DET in Q2) or a value (such as LOAD_VIDEO in Q2) as input, and return data in a tabular format. They are typically referenced in the FROM clause of a SELECT statement. On the other hand, scalar functions take values from a tuple as input, and return a single value or tuple, such as TimeOCR in Q1 (Lst. 1) and PosDet in Q2. They are commonly referenced in either the SELECT clause or the WHERE clause. Examples of how to create both types of UDFs are provided respectively in Lst. 5. Both examples employ externally referenced Python code incorporating CV models to fulfill the intended function objectives. While UDFs offer flexibility and functional expansion for handling tasks requiring ML/CV models, they can impact query performance, as UDFs might not be optimized in the same way as built-in functions of the database system [38].

Pipeline for executing a query in VDBMS. In VDBMS, a query is first parsed and interpreted to form a raw query plan, followed by an optimization phase to analyze and select the most effective plan [49]. Then, relevant video data is retrieved and processed according to the query requirements by executing the plan. In this paper, in contrast to many existing VDBMS that are limited to linear-structured query plans, we adapt VDBMS to accommodate tree-structured query plans. Queries with tree-structured plans may involve the join of results from ML models or other input sources



```
-- Q2: Retrieve running Tom Hanks with a car in the scene
      from Movie Forrest Gump.
WITH V AS LOAD_VIDEO('forrest_gump.mp4');
SELECT DISTINCT P.fid FROM PERSON_ID(V) AS P,
    OBJ_DET(V) AS D ON (P.fid = D.fid)
  WHERE P.label = 'Tom Hanks' AND D.label = 'car'
    AND PosDet(P.box) = 'running';
```

Listing 4: Example Query Q2.

```
CREATE SCALAR FUNCTION TimeOCR (img IMAGE, box BBOX)
  RETURN DATE
  AS RETURN IMPL 'sudfs/time_ocr.py';

CREATE TABLE FUNCTION OBJ_DET (V VIDEO)
  RETURN TABLE (
    id LONG, fid LONG FOREIGN KEY REFERENCES V.fid,
    label INT, box BBOX, scores FLOAT, PRIMARY KEY (id))
  AS RETURN IMPL 'tudfs/fastrcnn_object_detector.py';
```

Listing 5: Example UDFs. – Data types VIDEO denotes the table with schema (fid LONG, img IMAGE), IMAGE denotes NDARRAY UINT8(3, ANYDIM, ANYDIM), BBOX denotes NDARRAY FLOAT32(4).

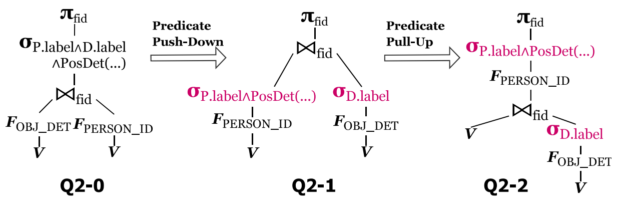


Figure 3: Query Plans of Q2. – Directly interpreted plan of Q2 and the plans after rule-based optimizations. The predicate filters (denoted as σ) are streamlined for clarity.

(e.g., joining visual, audio and text data). The raw query plan of Q2 interpreted by the parser is illustrated in Fig. 3 (Q2-0). It typically consists of the nodes of table UDFs (denoted as F), predicate filters (denoted as σ), along with other framework nodes such as joins and projections.

Query optimization in VDBMS. In VDBMS, the raw query plan first undergoes Rule-Based Optimization (RBO) by a conventional optimizer, employing both standard RDBMS rules such as predicate decomposition and push-down (e.g. Fig. 3 (Q2-1)) and video query-specific rules like predicate pull-up (e.g. Fig. 3 (Q2-2), which pulls up predicates from a branch of the join subtree) [19, 34, 45]. However, due to the challenges associated with querying video data (e.g. uncertainty introduced by ML models), RBO often falls short in effectively identifying the optimal query plan on its own. In contrast, Cost-Based Optimization (CBO) offers a more suitable approach. It begins with conducting a logical transformation and enumerating various *functionally equivalent* query plans, and then determines the best one by estimating and comparing the costs (typically estimated by table statistics or empirical methods) associated with each plan. Recent research has shed light on innovative approaches to generate more functionally equivalent query plans, including introducing proxies (i.e., lightweight filters) ahead of costly ML models [34], altering the processing order of different models [50], and introducing relational hints for the models [41].

3 CLUES

Clues are designed to convey user knowledge for query optimization. Our system, ClueVQS, comes equipped with five predefined Clue types proposed, each featuring its own optimization rules to be detailed in §3.1. These rules distill the abstract query-related knowledge encapsulated in Clues into executable optimization strategies that can rewrite query plans. Moreover, the rules are designed to be independent of any specific query or video content. As a result, as shown in Fig. 2, to process a specific query in ClueVQS, users only need to declare all available Clue instances, without concern about the optimization rules associated with each Clue type or the compatibility and effectiveness of each Clue instance for query optimization. ClueVQS will automatically parse all the declared Clue instances, identify those that can be effectively applied to the query, and apply the optimization rules of the corresponding Clue type to optimize the query (to be discussed in §4). Additionally, ClueVQS can easily accommodate new Clue types. The considerations and guidelines for creating new Clue types are detailed in §3.2. The way to declaring Clue instances is outlined in §3.3.

3.1 Supported Clue Types in ClueVQS

In this section, we introduce five distinct Clue types, including three based on novel optimization principles we propose (`MONOTONIC`, `DISJOINT`, `DOMAIN_MAPPING`), and two adapting ideas introduced in previous research (`PREFILTER`, `BULK_SUBSTITUTE`), thus demonstrating that our framework is general enough to encompass other types of optimization knowledge. Typically, Clues assist in optimizing queries by manipulating nodes within the query plan represented as a tree, such as adding new nodes or altering existing ones. We also detail their respective optimization rules in this section.

3.1.1 *Clue Type* `MONOTONIC`.

Clues of this type indicate that the output relation of a scalar UDF (or a column in the output relation of a table UDF), denoted as F_{Mono}, has a monotonic property in its values across the whole video or within a specified segment of the video. For instance, in query `Q1` (in Lst. 1), the outputs of UDF `TimeOCR` increase monotonically over the sequence of the video frames.

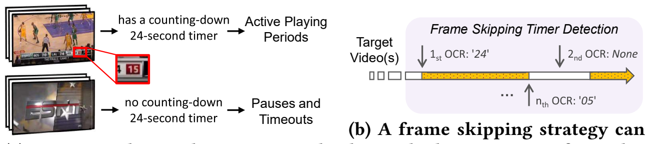
Optimization Rules. We can optimize query processing when applying the UDF F_{Mono} by skipping frames that do not satisfy the predicates without applying F_{Mono} to every frame, utilizing search algorithms such as binary search or interpolation search. For instance, as depicted in Fig. 1b, query `Q1` (in Lst. 1) is optimized using a binary search procedure when the variable in the predicate (i.e., the outputs of the scalar UDF `TimeOCR`) is monotonically increasing. The query efficiency is improved by applying `TimeOCR` only to the middle frame at each iteration of the binary search procedure.

3.1.2 *Clue Type* `DISJOINT`.

Unlike `MONOTONIC`, Clues in this new type provide information for patterns that repeat across *disjoint video segments*, which can be leveraged to optimize query processing. For instance, in query `Q3` (in Lst. 6), the video frames in a basketball game video satisfying the query predicate constitute disjoint video segments containing active playing periods. Fig. 4a reveals a Clue that a 24-second countdown timer appears on the screen in each disjoint video segment, initializing from 24 and then keeps decreasing. The detection of the 24-second timer helps separate segments containing active playing from pauses and timeouts as is typical in basketball games.

```
-- Q3: Retrieve active playing period from a basketball
     game video.
SELECT * FROM LOAD_VIDEO('basketball_game.mp4')
  WHERE ActivePlayingDet(img) = TRUE;
```

Listing 6: Example Query Q3.



(a) A 24-second countdown timer is always visible during active play. **(b) A frame skipping strategy can be applied to minimize frame-by-frame processing.**

Figure 4: Optimization of Q3 utilizing the Clue that a 24-second countdown timer is always visible during active play.

```
-- Q4: Retrieve the cars turning left at the intersection
      from a surveillance video.
SELECT T.tid FROM OBJ_TRACK(LOAD_VIDEO(
    'surveillance.mp4')) AS T
  WHERE T.obj = 'car' AND T.boxes.x1 >= 500 AND T.boxes.
    y1 >= 200 AND Trajectory(T.boxes) = 'turning left';
```

Listing 7: Example Query Q4.

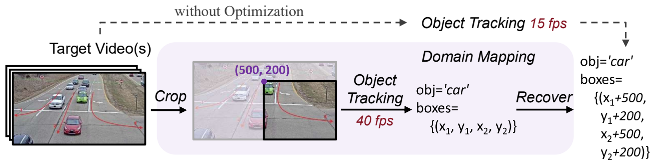


Figure 5: In the example of Q4, Clue `DOMAIN_MAPPING` applies object tracking models solely on cropped images to reduce inference overhead.

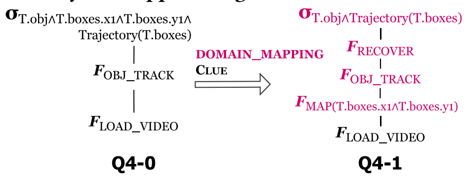

σT.obj∧T.boxes.x1∧T.boxes.y1∧Trajectory(T.boxes)
FOBJ_TRACK
FLOAD_VIDEO
DOMAIN_MAPPING
Clue
σT.obj∧Trajectory(T.boxes)
FRECOVER
FOBJ_TRACK
FMAP(T.boxes.x1∧T.boxes.y1)
FLOAD_VIDEO
Q4-0
Q4-1

Figure 6: Query Plans of Q4 Revised by Clues in Type `DOMAIN_MAPPING`.

Optimization Rules. It can optimize query processing by leveraging such repeating patterns within the disjoint video segments, similar to the values of the 24-second timer (which always start at 24 and decreases monotonically in each segment), as illustrated in Fig. 4b More optimization strategies can be developed with a richer set of query-related knowledge provided by Clue `DISJOINT`.

3.1.3 *Clue Type* `DOMAIN_MAPPING`.

Fig. 5 illustrates an optimization applied to the UDF `OBJ_TRACK` (object tracking model [6, 7]) of query `Q4` (in Lst. 7), including two main steps: (1) cropping the image according to bounding box predicates (i.e., removing areas of irrelevant pixels that the predicates are not concerned with), and (2) mapping the bounding boxes from the cropped image back to the original (i.e., applying offsets based on the cropped pixels). We generalize such optimization and use `DOMAIN_MAPPING` to denote such Clue types.

Optimization Rules. Clues in this type can be applied to rewrite the original query by adding two new UDFs, denoted as F_{MAP} and F_{RECOVER}, positioned before and after an existing UDF (e.g. $F_{\text{OBJ\_TRACK}}$), respectively, and pushing down some of its predicates into F_{MAP} (e.g. `boxes.x1 >= 500 AND boxes.y1 >= 200`), as demonstrated in Fig. 6. This allows F_{MAP} to map the input relation of $F_{\text{OBJ\_TRACK}}$ to a relation that is easier to process for $F_{\text{OBJ\_TRACK}}$.

```
-- Q5: Retrieve a specific event from a BBC documentary.
SELECT * FROM EVENT_DET(LOAD_VIDEO('bbc_earth.mp4'))
  WHERE event = 'wildebeests attacked by crocodiles';
```

Listing 8: Example Query Q5.

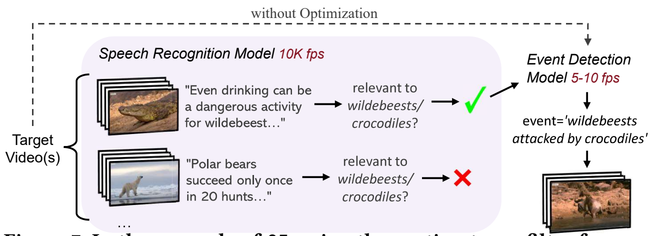

without Optimization
Speech Recognition Model 10K fps
"Even drinking can be a dangerous activity for wildebeest..."
relevant to wildebeests/ crocodiles?
"Polar bears succeed only once in 20 hunts..."
relevant to wildebeests/ crocodiles?
Target Video(s)
Event Detection Model 5-10 fps
event='wildebeests attacked by crocodiles'

Figure 7: In the example of Q5, using the caption to prefilter frames that are not related to the queried event can prevent the need to apply the computationally intensive model to every frame.

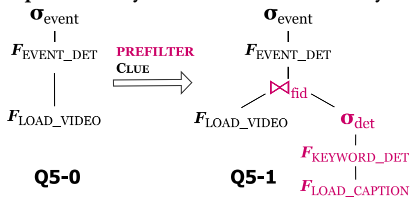


Figure 8: Query Plans of Q5 revised by CLUES in Type PREFILTER.

Subsequently, F_{RECOVER} is used to remap the outcomes processed by $F_{\text{OBJ\_TRACK}}$ back to the original results.

3.1.4 CLUE Type PREFILTER. A very common optimization technique in VDBMS is to use prefilters to pre-process certain frames and only send promising ones for further evaluation to computationally intensive ML models [10, 25, 34, 41]. In line with this approach, we demonstrate how prefilters can be expressed in our framework and also enable them to incorporate diverse sources when using CLUE instances. For example, for query Q5 (in Lst. 8) that aims to locate a specific event within a lengthy documentary, the caption recognized by a speech model (10k fps[4]) [35] can be utilized to prefilter frames with caption irrelevant to the queried event, thereby avoiding the application of the computationally expensive event detection model (5-10 fps) at each frame, as illustrated in Fig. 7.

Optimization Rules. CLUES in this type can be applied to rewrite the original query plan by introducing a prefilter as a new branch, positioning it as a child of the filtered ML model in the plan. For example, Fig. 8 illustrates the plan after applying the CLUE based on the knowledge of the video caption illustrated in Fig. 7 on query Q5 (in Lst. 8). The output relation of the prefilter should align with one or more common columns (e.g., fid) of the filtered ML model input. While executing the query plan revised by CLUES in this type, the prefilter applies its predicate (e.g. σ_{det}) to filter entries from the input of the filtered ML model (e.g. $F_{\text{EVENT\_DET}}$), thereby reducing the computational overhead of the latter.

3.1.5 CLUE Type BULK_SUBSTITUTE. Another common optimization technique similar to prefilter is to substitute one model with another that may be more time-efficient or accurate [41, 50]. We demonstrate how this can be adapted in our framework and also extend it to allow for the substitution of multiple models, not just one. The models involved are not restricted to being functionally similar.

[4]Frames per second (fps) of a model refers to the number of frames it can process or analyze per second.

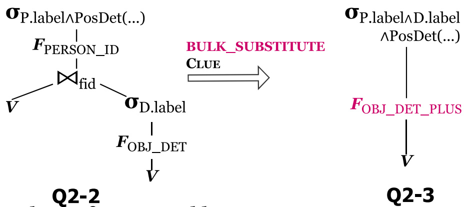


Figure 9: Plans of Q2 revised by CLUES in Type BULK_SUBSTITUTE.

For instance, for query Q2 (in Lst. 4), a more versatile model, such as OBJ_DET_PLUS, which is capable of identifying both persons (detected by PERSON_ID) and objects (detected by OBJ_DET), can replace two original models, as demonstrated in Fig. 9.

Optimization Rules. CLUES in this type can rewrite the original query by introducing a new UDF, referred to as F_S, to substitute one or more previously used UDFs. F_S produces an output relation covering columns (e.g., label and box) in the predicate filters of the UDFs being replaced. A new predicate filter for F_S is created combining the predicate filters of the substituted UDFs.

3.2 Defining New CLUE Types

CLUEVQS, is designed to easily accommodate new CLUE types, as the example presented in Lst. 2. To create a new CLUE type, initially, we define the name and argument list for this new type, along with the data types and any constraints. Following this, we develop a new optimization rule for this CLUE type, which could involve rewriting subtrees of the original query plan or improving the execution of certain nodes or subtrees within the plan.

The argument list provided to a CLUE type brings two benefits: (1) for each CLUE type, developers can pre-define all arguments for important parameters to be used in developing optimization rules for CLUE instances; (2) exposing the parameters to end-users could significantly improve the flexibility and applicability of using CLUE instances, allowing users to provide insights bounded to specific application domains or video sources. Such arguments are used to instantiate CLUE types with specific query information. In CLUEVQS, we provide an API for implementing these rules [9]. The rule implemented for a new CLUE type will be used by the optimizer CGen (to be discussed in §4.1), enabling CLUEVQS to rewrite the query plan accordingly.

3.3 Declaration of CLUE Instances

CLUE instances are declared right after the query statement using a simple and straightforward syntax of key-value pairs, making it easy for users to implement. Lst. 3 presents how CLUE instances for query Q1 (in Lst. 1) are specified. The CLUE type (specified with clue_type) is mandatory for all CLUE instance declarations. The key-value pairs vary depending on the CLUE type, and are determined based on the query itself: monotonic_on key indicates the UDF with monotonic outputs; decreasing specifies whether the monotonicity is increasing or decreasing; nullable denotes if the UDF results can be null. As shown in Lst. 3, users simply fill in the values corresponding to these keys to reflect their knowledge about monotonicity, without concerns about compatibility and effectiveness.

4 CLUE-BASED OPTIMIZATION IN CLUEVQS

Although §3 presents numerous CLUES, utilizing them effectively to optimize queries remains a challenge. This section proposes the cost-based optimization framework adopted by CLUEVQS. In §4.1,

Algorithm 1: Plan Generation with CLUES

```
Procedure CGen($q^o$, $\mathbb{C}$)
    $\mathbb{G}_D$ ← GraphBuilder($\mathbb{C}$) ;
    $_{ord}V_D$ ← TopologicalSort($\mathbb{G}_D$) ;
    $\mathbb{Q}^+ \leftarrow \emptyset$ ;
    PlanGeneratorRecursion($_{ord}V_D$, $\emptyset$) ;
    OrderPermutation($\mathbb{Q}^+$) ;
    return $\mathbb{Q}^+$ ;
Procedure PlanGeneratorRecursion($_{ord}V'$, $V_{use}$)
    $\xi \leftarrow {}_{ord}V'$.pop() ;
    if $_{ord}V' \neq \emptyset$ then
        PlanGeneratorRecursion($_{ord}V'$, $V_{use}$) ;
        if ClueVerifier($\xi$, $V_{use}$) then
            PlanGeneratorRecursion($_{ord}V'$, $V_{use} \cup \{\xi\}$) ;
    if $_{ord}V' = \emptyset \wedge V_{use} \neq \emptyset$ then
        $\mathbb{Q}^+ \leftarrow \mathbb{Q}^+ \cup \{$PlanRewriter($q^o$, $V_{use}$)$\}$ ;
```

we introduce a plan generator, CGen, to apply the CLUES strategically and enumerate all candidate query plans. §4.2 formalizes the problem of identifying the optimal plan from all candidate query plans, known as *MinETOpt*. We next introduce a plan selector, CSel, employed by CLUEVQS as a solution to the problem *MinETOpt*. Finally, §4.4 discusses a variant of the aforementioned problem that incorporates accuracy constraints, namely *MinETOpt-AC*, along with a brief overview of how our proposals adapt to this new problem (referred to as CSel-AC) and its limitations.

4.1 CGEN

As shown in Fig. 2, CLUEVQS accepts a query along with a collection of CLUES, denoted as $\mathbb{C}$, as inputs. Initially, CLUEVQS optimizes the query through a conventional rule-based query optimizer (as described in §2) to generate a query plan q^o. Then, with the original query plan q^o and user-provided CLUES $\mathbb{C}$ as inputs, the algorithm, CGen, is employed to generate all candidate query plans $\mathbb{Q}^+$, as outlined in Alg. 1. Through a recursive procedure, CGen iteratively visits all CLUES and then rewrites the query plans to catalogue all candidate query plans.

At the outset, CGen constructs a graph to represent dependencies among the CLUES, utilizing GraphBuilder, as outlined in Lines 2-3 of Alg. 1. The dependencies play a crucial role in the CLUE validation process (to be elaborated in §4.1.1). Let $\mathbb{G}_D$:$\{V_D, E_D\}$ represent the dependency graph, where each vertex is a CLUE, and each directed edge $u \rightarrow v$ signifies that v depends on u. We denote the topological sort of all its vertices as $_{ord}V_D$. TopologicalSort can compute $_{ord}V_D$ for $\mathbb{G}_D$ efficiently in linear time by employing methods such as Kosaraju's algorithm [17].

Subsequently, in Line 5 of Alg. 1, CGen visits all the CLUES and generates all candidate query plans using a recursive approach called PlanGeneratorRecursion. It accepts two parameters: one is the queue of CLUES (in topological order) that have yet to be visited, denoted as $_{ord}V'$, and the other is the collection of CLUES used for generating plans, denoted as V_{use}. During each recursion of PlanGeneratorRecursion, a CLUE ξ within the queue $_{ord}V'$ will be visited, and its applicability will be verified (Line 12; explained in §4.1.1): if its verification fails, the algorithm will only enter the branch with the same parameter as V_{use} (Line 11); however, if it is verified, the algorithm will additionally enter the branch with $V_{use} \cup \{\xi\}$ (Line 13). When the unvisited queue $_{ord}V'$ becomes empty while V_{use} is not an empty set, the recursion will halt, leading to the generation of a new plan (Lines 14-15; elaborated in §4.1.2). After all the recursive steps halt, $\mathbb{Q}^+$ will have contained all the candidate plans generated throughout the recursion.

In addition, CGen can also take into account other query rewriting techniques, such as predicate pull-up (illustrated in Fig. 3) and predicate order permutation (elaborated in §4.1.3).

4.1.1 CLUE Validation. In Line 12 of Alg. 1, ClueVerifier is utilized to assess whether a CLUE ($\in\mathbb{C}$) is valid for the queries. For each type of CLUES, the validator will employ a dedicated built-in fast static procedure to verify its applicability to the current query plan. This may involve verifying the presence of arguments in the declaration and assessing if the optimization rule is compatible with the query. We exemplify the procedure of CLUE validation using the CLUE Q1_CLUE1 in type of MONOTONIC declared in Lst. 3: the validator will verify if the attribute monotonic_on is present in the query predicates and verify its data type. Some CLUES may depend on others, where these dependencies originate from their associated optimization rules. For example, when a CLUE needs to use UDFs introduced by other CLUES, it can only be applied after the other CLUES have been applied.

4.1.2 Rewriting a Query Plan. As per Line 15 of Alg. 1, PlanRewriter is utilized to create a new query plan by rewriting the original query plan (q^o) utilizing a given set of CLUES (V_{use}). PlanRewriter sequentially applies each CLUE ($\in V_{use}$) in a topological order:

- ▹ For each $h_i \in V_{use}$, it applies this h_i to rewrite the query plan generated by the previous h_{i-1} (i.e. $q^{(i-1)}$; the first h_1 will rewrite q^o), creating a new query plan $q^{(i)}$, by adopting its optimization rule appropriately (as detailed in §3.1).

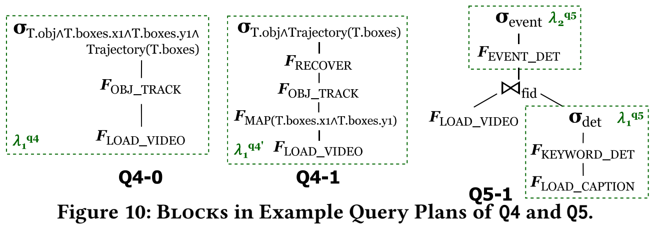
4.1.3 Order Permutation. As per Line 6 of Alg. 1, for every generated query plan in $\mathbb{Q}^+$, OrderPermutation permutes the order of predicates in each of its predicate filters, by placing the predicates without scalar UDFs at the beginning and exhausting all possible orderings of the remaining predicates.

4.2 Optimization Problem Formulation

After CGen generates all candidate query plans $\mathbb{Q}^+$, CLUEVQS aims to select a query plan, denoted as q^* ($\in\mathbb{Q}^+$), which has the lowest execution time under the same input setting compared to all other plans.

4.2.1 Canary Data. CLUEVQS utilizes a canary input ($\mathbb{D}_C$) to assess the cost of each candidate plan. The canary input is a user-specified representative sample of the input being queried, such as a short clip from the video involved in the original query. Using small, representative canaries is common for tuning and calibration in computer vision, machine learning, and data mining systems before execution [33, 41]. They have also been used by recent VDBMS [41, 48–50]. Our empirical evaluation proves that it is sufficient to use a canary with at least one instance of each CLUE proposed by users for optimization, to assess the performance of the CLUES.

4.2.2 MinETOpt: Problem Statement. For each plan $q \in \mathbb{Q}^+$, let $\widehat{C}(q)$ represent the execution cost of q on the canary input. CLUEVQS aims to select the optimal plan q^* that minimizes the execution cost $\widehat{C}(q^*)$, by strategically evaluating the performance of all plans on the canary input. Meanwhile, we should keep this evaluation cost



$\sigma_{\text{T.obj}\wedge\text{T.boxes.x1}\wedge\text{T.boxes.y1}\wedge\text{Trajectory(T.boxes)}}$
$F_{\text{OBJ\_TRACK}}$
$F_{\text{LOAD\_VIDEO}}$
λ_1^{q4}
Q4-0
$\sigma_{\text{T.obj}\wedge\text{Trajectory(T.boxes)}}$
F_{RECOVER}
$F_{\text{OBJ\_TRACK}}$
$F_{\text{MAP(T.boxes.x1}\wedge\text{T.boxes.y1)}}$
$\lambda_1^{q4'}$
$F_{\text{LOAD\_VIDEO}}$
Q4-1
σ_{event} λ_2^{q5}
$F_{\text{EVENT\_DET}}$
$\bowtie_{\text{fid}}$
$F_{\text{LOAD\_VIDEO}}$
σ_{det} λ_1^{q5}
$F_{\text{KEYWORD\_DET}}$
$F_{\text{LOAD\_CAPTION}}$
Q5-1

Figure 10: Blocks in Example Query Plans of Q4 and Q5.

as small as possible. Let $E(q|q^*)$ represent the cost of evaluating each plan q ($\in\mathbb{Q}^+$) on the canary input to identify q^*. Formally, the problem of finding the optimal plan is defined as:

$$q^* \coloneqq \arg\min_{q\in\mathbb{Q}^+} \widehat{C}(q). \tag{1}$$

The goal of the optimization problem, *MinETOpt*, is to:

$$\text{minimize} \sum_{q\in\mathbb{Q}^+} E(q|q^*). \tag{2}$$

4.2.3 Naive Approach. A naive approach to the problem *MinETOpt* is to execute each plan $q\in\mathbb{Q}^+$ on $\mathbb{D}_C$ to evaluate their performance, whose total evaluation cost is $E_{\max}=\sum_{q\in\mathbb{Q}^+}\widehat{C}(q)$. However, as the number of Clues ($|\mathbb{C}^+|$) grows, the search space for candidate query plans ($|\mathbb{Q}^+|$) expands exponentially, making this approach impractical. Therefore, it is imperative to devise an algorithm capable of efficiently identifying the optimal plan, avoiding assessing the candidate plans one by one. In §4.3, we will propose an algorithm, CSel, that strategically allocates evaluation resources and significantly reduces the evaluation cost.

4.3 CSel

We propose an algorithm, namely CSel, to address *MinETOpt*, selecting the optimal query plan (q^*) from a collection of candidate query plans ($\mathbb{Q}^+$) efficiently. The algorithm follows a branch-and-bound approach [2, 49], assigning upper and lower bounds to each query plan. As queries are progressively evaluated on the canary data, these bounds are updated, which enables CSel to prune plans safely, effectively narrowing down the number of plans to be completely evaluated, thereby accelerating optimal plan selection.

4.3.1 Preliminaries. In a query plan tree, a structure consisting of a predicate filter followed by several related table UDFs is referred to as a Block, denoted as λ, as illustrated in Fig. 10. A plan tree consists of one or more Blocks, which we sequentially order based on their distance from the leaf nodes, such as the λ_1^{q5} and λ_2^{q5} of Q5-1 in Fig. 10. Specifically, a plan q includes n^q Blocks: λ_1^q, ..., $\lambda_{n^q}^q$, where a smaller index number indicates a lower level and a smaller subtree the Block is at. We assume that the processing cost of a Block for one row from the input relation, such as a single video frame, remains constant. Let $c(\lambda)$ represent the processing costs of a Block λ on a single row. Thus, the execution cost of a Block depends only on its input row count. Let $\hat{v}(\lambda_i^q)$ represent the cardinality of the outputs of the Block λ_i^q when evaluating q on the canary. The cost of the Block λ_i^q when evaluating q on the canary is:

$$\widehat{C}(\lambda_i^q) = \hat{v}(\lambda_{i-1}^q)\cdot c(\lambda_i^q), \tag{3}$$

Algorithm 2: Cost-Based Plan Selection

```
1  Procedure CSel($\mathbb{D}_C$, $\mathbb{Q}^+$)
2      $\mathbb{B} \leftarrow$ GroupBlock($\mathbb{Q}^+$) ;
3      bounds $\leftarrow$ BoundInitializer($\mathbb{B}$) ;
4      $\mathbb{Q}_{\text{pruned}} \leftarrow \emptyset$ ;
5      while $\neg$ HaltCondition do
6          $\lambda' \leftarrow$ TopUnexplored($\mathbb{B}$, $\mathbb{Q}_{\text{pruned}}$) ;
7          BlockEvaluate($\mathbb{D}_C$, $\lambda'$) ;
8          BoundUpdate($\lambda'$) ;
9          $\mathbb{Q}_{\text{pruned}} \leftarrow$ PlanPruner($\mathbb{Q}^+$, bounds, $\mathbb{Q}_{\text{pruned}}$) ;
10     return $q^*$ ;
```

where λ_{i-1}^q is the preceding Block of λ_i^q. Clearly, the cost of evaluating a query q on the canary is

$$\widehat{C}(q) = \sum_{i=1}^{n^q}\widehat{C}(\lambda_i^q). \tag{4}$$

The Blocks present in different query plans could be identical, or they might be related, such as be output-equivalent, namely they produce the same outputs for identical inputs (e.g. $\lambda_1^{q4}\sim\lambda_1^{q4'}$ in Fig. 10). Let $O(\lambda|I)$ represent the outputs generated by executing Block λ with I as the inputs. We will leverage these characteristics of Blocks to avoid the full evaluation of each plan, thereby reducing the evaluation cost.

4.3.2 Algorithm. CSel utilizes an iterative method, exploring (executing) an unexplored Block of a plan at each iteration, to update the bounds of plans and prune plans, as detailed in Alg. 2.

At the beginning (Line 2 of Alg. 2), CSel enumerates all the Blocks in $\mathbb{Q}^+$, denoted as $\mathbb{B}$, and identifies the identical or output-equivalent ones. Following that, as per Line 3 of Alg. 2, CSel initializes the upper and lower bounds of the cost of all queries in $\mathbb{Q}^+$, denoted as $\widehat{C}_{\text{ub}}(q)$ and $\widehat{C}_{\text{lb}}(q)$, and the bounds on the cardinality for all Blocks in $\mathbb{B}$, denoted as $\hat{v}_{\text{ub}}(\lambda)$ and $\hat{v}_{\text{lb}}(\lambda)$, according to Lemma 4.1 (elaborated in §4.3.3).

In Lines 5-9 of Alg. 2, CSel iteratively performs the following steps: it selects and executes an unexplored Block λ', updates the cost bounds, and prunes plans. The iteration repeats until the HaltCondition is triggered, i.e. there exists an optimal plan whose cost upper bound is lower than the costs of all other plans:

$$\texttt{HaltCondition}: \quad \exists\, q^*,\ \forall_{q_i\in\mathbb{Q}^+}\ \widehat{C}_{\text{ub}}(q^*) \le \widehat{C}_{\text{lb}}(q_i).$$

As per Line 6, TopUnexplored ranks all plans by the average of their upper and lower cost bounds, calculated as $\widehat{C}_{\text{avg}}(q) = \frac{\widehat{C}_{\text{ub}}(q)+\widehat{C}_{\text{lb}}(q)}{2}$, and selects an unexplored Block from the plan with the smallest average cost, namely λ'. Subsequently, as per Line 7, BlockEvaluate executes the subtree of the query plan located at Block λ' on the canary. Since all other Blocks within this subtree have already been executed in previous iterations, the cost of executing this subtree equates to the cost of executing Block λ'. Following that, in Line 8, CSel updates the bounds for λ' and for all the plans relevant to λ'. Lastly, as per Line 9, CSel prunes plans by comparing the cost bounds of each pair of plans (e.g. q_i and q_j) to eliminate those that cannot be optimal, specifically, if a plan's lower bound exceeds the upper bound of another plan,

$$\forall_{q_i,q_j\in\mathbb{Q}^+}\ \widehat{C}_{\text{ub}}(q_i) \le \widehat{C}_{\text{lb}}(q_j) \quad\Rightarrow\quad \text{prune } q_j.$$

4.3.3 Upper and Lower Cost Bounds. As indicated by Eq. 3, the estimation of the upper and lower bounds for the cost of a Block λ_i (i.e. $\widehat{C}_{\text{ub}}(\lambda_i)$ and $\widehat{C}_{\text{lb}}(\lambda_i)$) essentially involves estimating the upper and lower bounds for the cardinality of its preceding Block λ_{i-1} (i.e. $\hat{v}_{\text{ub}}(\lambda_{i-1})$ and $\hat{v}_{\text{lb}}(\lambda_{i-1})$).

$$\widehat{C}_{\text{ub}}(\lambda_i) = \hat{v}_{\text{ub}}(\lambda_{i-1}) \cdot c(\lambda_i); \quad \widehat{C}_{\text{lb}}(\lambda_i) = \hat{v}_{\text{lb}}(\lambda_{i-1}) \cdot c(\lambda_i). \tag{5}$$

The upper bound of the cardinality for a Block λ_i is typically a fraction (or multiple) of the input row count,

$$\hat{v}_{\text{ub}}(\lambda_i) = \mu(\lambda_i) \cdot \hat{v}_{\text{ub}}(\lambda_{i-1}), \tag{6}$$

where $\mu(\lambda_i)$ is the largest fraction (or multiple) of input rows that Block λ_i can output, a value influenced by the inherent properties of the UDFs in the Block and established by expert guidance. The lower bound of the cardinality for a Block λ_i is $\hat{v}_{\text{lb}}(\lambda_i)=0$. Eventually, the upper and lower bounds for the cost of a query plan q (i.e. $\widehat{C}_{\text{ub}}(q)$ and $\widehat{C}_{\text{lb}}(q)$) can be estimated by summing the upper and lower bounds for the costs associated with all its Blocks.

$$\widehat{C}_{\text{ub}}(q) = \sum_{j=1}^{n^q} \widehat{C}_{\text{ub}}(\lambda_j^q); \quad \widehat{C}_{\text{lb}}(q) = \sum_{j=1}^{n^q} \widehat{C}_{\text{lb}}(\lambda_j^q). \tag{7}$$

`BoundInitializer`: During the initialization (Line 3 of Alg. 2), without additional information, the lower bound for a Block's output cardinality is set to 0, and its upper bound is set based on the cardinality of the preceding Blocks.

Lemma 4.1. *Cost Bound Initialization. For each plan, the initialized upper and lower bounds of the cardinality for the output of a Block λ_i are set as follows: $\hat{v}_{ub}(\lambda_i)=\mu(\lambda_i) \cdot \hat{v}_{ub}(\lambda_{i-1})$, $\hat{v}_{lb}(\lambda_i)=0$, where $\hat{v}_{ub}(\lambda_0)=\hat{v}_{lb}(\lambda_0)=|\mathbb{D}_C|$, the row count of the canary inputs. The initialized upper and lower bounds of the cost for a plan q are set as follows: $\widehat{C}_{ub}(q) = \sum_{j=1}^{n^q} \hat{v}_{ub}(\lambda_{j-1}^q) \cdot c(\lambda_j^q)$, $\widehat{C}_{lb}(q) = \sum_{j=1}^{n^q} \hat{v}_{lb}(\lambda_{j-1}^q) \cdot c(\lambda_j^q)$.*

`BoundUpdate`: At each iteration, after executing a Block λ', the upper and lower bounds of its cost and output cardinality are both updated to a single value, i.e. $\widehat{C}(\lambda')$ and $\hat{v}(\lambda')$. The cost bounds of the plan λ' belongs to can be directly updated utilizing Eq. (5), Eq. (6) and Eq. (7).

4.3.4 Example. We illustrate Alg. 2 with an example as shown in Tb. 1. For simplicity, we consider only three query plans from a set, namely q_1, q_2, and q_3. Tb. 1a illustrates the relationship between Blocks produced by `GroupBlock`: $\lambda_2^{q_1}$ and $\lambda_3^{q_3}$ are output-equivalent; $\lambda_2^{q_2}$ and $\lambda_3^{q_3}$ are identical. Tb. 1b presents the estimated cost bounds and the explored Blocks of the three plans, after $(i-1)$-th iteration.

The input for the front-most unexplored Block in each plan (i.e. $\lambda_2^{q_1}$, $\lambda_1^{q_2}$ and $\lambda_3^{q_3}$) is the output of its preceding node, which has been obtained in prior iterations. For instance, the inputs for $\lambda_2^{q_1}$ and $\lambda_3^{q_3}$ are $O(\lambda_1^{q_1})$ and $O(\lambda_2^{q_3})$ respectively; meanwhile, $\lambda_1^{q_2}$, which lacks a preceding node, receives its input from canary data $\mathbb{D}_C$.

At i-th iteration, `CSel` uses `TopUnexplored` to rank these plans based on their $\widehat{C}_{\text{avg}}(q)$ and selects the one with the smallest $\widehat{C}_{\text{avg}}(q)$, which is q_3. It then uses `BlockEvaluate` to explore the front-most unexplored Block in q_3 (i.e. $\lambda'=\lambda_3^{q_3}$). This involves executing $\lambda_3^{q_3}$ on $O(\lambda_2^{q_3})$, which yields the outcome $O(\lambda_3^{q_3})$ and the associated cost $\widehat{C}(\lambda_3^{q_3})$. Then, `BoundUpdate` updates the bounds of $\lambda_3^{q_3}$ accordingly. Other Blocks that have a relationship with $\lambda_3^{q_3}$ (as indicated in Tb. 1a) will have their bounds updated by `CSel` without actual execution. This is achieved by utilizing the selectivity of $\lambda_3^{q_3}$ on its input $O(\lambda_2^{q_3})$. Since $\lambda_2^{q_1}$ and $\lambda_2^{q_2}$ are expected to produce the same outcomes as $\lambda_3^{q_3}$ when provided with the same inputs, it is possible to update their bounds by computing the overlap between their inputs with that of λ'.

After updating the upper and lower bounds of the cost for each plan, as illustrated in Tb. 1c, at the end of the iteration, `CSel` uses `PlanPruner` to prune plans whose lower bound exceeds the upper bounds of other plans (e.g., the pruning of q_3).

4.4 Accuracy Constraint

Since machine learning models are prone to prediction errors, incorporating these models into certain Clue instances might reduce the accuracy of query results. The query plan optimized by Clues generated by `CGen` might exhibit reduced accuracy compared to the original query plan due to these errors. Instead of relying on the user to assess the accuracy impact of Clues, the system ClueVQS is supposed to automatically eliminate plans that fall below a certain accuracy threshold. This is achieved by introducing an accuracy constraint (user provided), which specifies the maximum acceptable reduction in accuracy resulting from the application of Clues. We update the aforementioned problem formulation in §4.4.1 to account for such a potential reduction in accuracy. Additionally, we discuss a method adapted from `CSel`, namely `CSel-AC`, that addresses this issue and its limitations in §4.4.2, hoping to inspire future research.

4.4.1 MinETOpt-AC: Problem Statement with Accuracy Constraint. We adopt the query-wide accuracy, A ($\in[0,1]$) [34, 50], to measure the accuracy of a plan compared to the original query plan q^o. Let α represent the minimum tolerant accuracy. The problem of identifying the optimal plan under this setting becomes:

$$q^*_{AC} \coloneqq \arg\min_{q\in\mathbb{Q}^+} \widehat{C}(q), \text{s.t. } \widehat{A}(q) \geq \alpha, \tag{8}$$

The goal of the new optimization problem with accuracy constraint, named *MinETOpt-AC*, becomes:

$$\text{minimize} \sum_{q\in\mathbb{Q}^+} E(q|q^*_{AC}). \tag{9}$$

A widely used method for calculating query-wide accuracy in other VDBMS [34, 50] involves using the *recall rate*[5] of the outputs produced by the revised plan compared to the ground truth outputs (or the outputs produced by the original plan if no ground truth is available). A way to estimate this accuracy is to continue using a canary. Specifically, the query-wide accuracy $\widehat{A}(q)$ of a query q on the canary is the recall rate of the row-level[6] outputs produced by q (denoted as $\widehat{O}(q)$) versus the ground truth outputs (denoted as $\widehat{O}_{\text{GT}}(q)$),

$$\widehat{A}(q) = \frac{|\widehat{O}(q) \cap \widehat{O}_{\text{GT}}(q)|}{|\widehat{O}_{\text{GT}}(q)|}. \tag{10}$$

If the canary has no ground truth, an alternative method to compute the query-wide accuracy is: $\widehat{A}(q) = \frac{|\widehat{O}(q)\cap\widehat{O}(q^o)|}{|\widehat{O}(q^o)|}$, where $\widehat{O}(q^o)$

[5]Our approach aligns with prior works in defining the optimization problem with recall, because, while optimizations often lead to efficiency gains, they typically reduce recall by potentially missing relevant results, yet do not necessarily affect precision.

[6]The row-level outputs can not only refer to the rows of the input table (e.g., when querying `fid`, each row corresponds to a frame), or refer to the rows of a table produced by a table UDF (e.g., the table generated by `OBJ_TRACK` as per Q4 in Lst. 7, where each row corresponds to a tracking result).

Table 1: Example of CSel Handling an Iteration.

(a) Relationship between Blocks.

$\mathbb{B}$
$\lambda_2^{q_1} \sim \lambda_3^{q_3}$
$\lambda_2^{q_2} \equiv \lambda_3^{q_3}$
...

(b) The state after the $(i-1)$-th iteration.

Plans	Blocks		$(\widehat{C}_{\text{lb}}(q), \widehat{C}_{\text{ub}}(q))$	$\widehat{C}_{\text{avg}}(q)$
	Explored	Unexplored		
q_1	$\lambda_1^{q_1}$	$\lambda_2^{q_1}, \lambda_3^{q_1}, ...$	(100, 200)	150
q_2	/	$\lambda_1^{q_2}, \lambda_2^{q_2}, ...$	(80, 160)	120
q_3	$\lambda_1^{q_3}, \lambda_2^{q_3}$	$\lambda_3^{q_3}, \lambda_4^{q_3}, ...$	(90, 110)	100

(c) The state after the i-th iteration.

Plans	Blocks		$(\widehat{C}_{\text{lb}}(q), \widehat{C}_{\text{ub}}(q))$	$\widehat{C}_{\text{avg}}(q)$
	Explored	Unexplored		
q_1	$\lambda_1^{q_1}$	$\lambda_2^{q_1}, \lambda_3^{q_1}, ...$	(100, $\underline{140}$)	$\underline{120}$
q_2	/	$\lambda_1^{q_2}, \lambda_2^{q_2}, ...$	(80, $\underline{100}$)	$\underline{90}$
q_3	$\lambda_1^{q_3}, \lambda_2^{q_3}, \underline{\lambda_3^{q_3}}$	$\lambda_4^{q_3}, ...$	(90, $\underline{100}$)	$\underline{95}$

Algorithm 3: Plan Selection with Accuracy Constraint

```
1  Procedure CSel-AC($\mathbb{D}_C$, $\mathbb{Q}^+$, $\alpha$)
2      $\mathbb{B} \leftarrow$ GroupBlock($\mathbb{Q}^+$) ;
3      bounds $\leftarrow$ **BoundInitializer-AC**($\mathbb{B}$) ;
4      $\mathbb{Q}_{\text{pruned}}, \mathbb{Q}_{\text{disqualified}} \leftarrow \emptyset, \emptyset$ ;
5      while $\neg$ HaltCondition do
6          $\lambda' \leftarrow$ TopUnexplored($\mathbb{B}, \mathbb{Q}_{\text{pruned}}, \mathbb{Q}_{\text{disqualified}}$) ;
7          BlockEvaluate($\mathbb{D}_C, \lambda'$) ;
8          **BoundUpdate-AC**($\lambda'$) ;
9          $\mathbb{Q}_{\text{disqualified}} \leftarrow$ **PlanExcluder-AC**($\mathbb{Q}^+$, bounds) ;
10         $\mathbb{Q}_{\text{pruned}} \leftarrow$ **PlanPruner-AC**($\mathbb{Q}^+$, bounds) ;
11     return $q^*$ ;
```

represents the outputs produced by the original plan q^o without Clue optimization.

4.4.2 CSel-AC. The previously proposed Alg. 2 requires only a few modifications to adapt to the new problem *MinETOpt-AC*, which involves estimating the accuracy of each plan and pruning those that fail to meet the accuracy constraint. This revised approach is referred to as CSel-AC, presented in Alg. 3.

Beyond cost, CSel-AC also estimates the upper bound for the accuracy of each plan on the canary, denoted as $\widehat{A}_{\text{ub}}(q)$,

$$\widehat{A}_{\text{ub}}(q) = \min\left(\frac{\hat{v}_{\text{ub}}(q)}{|\widehat{O}_{\text{GT}}(q)|}, 1.0\right), \tag{11}$$

where $\widehat{O}_{\text{GT}}(q)$ is the ground truth outputs of the canary. If the canary has no ground truth available, then the outputs generated by the original plan q^o can be used instead, i.e., $\widehat{O}_{\text{GT}}(q)$=$\widehat{O}(q^o)$.

CSel-AC also employs an iterative method to prune plans, but it modifies several functions, as highlighted in bold in Alg. 3. During the initialization, for each plan, BoundInitializer-AC additionally initializes the upper bound for the accuracy of each plan on the canary, using Eq. 11. In each iteration, after executing a Block, BoundUpdate-AC additionally updates the accuracy bounds ($\widehat{A}_{\text{ub}}(q)$) using the updated cardinality bounds through Eq. 11. PlanExcluder-AC excludes the plans whose accuracy upper bounds fall below the minimum accuracy constraint, specifically when $\widehat{A}_{\text{ub}}(q)$<$\alpha$. The halt condition HaltCondition becomes

$$\texttt{HaltCondition}: \quad \exists\, q^*, \ \forall_{q_i \in \mathbb{Q}^+ \setminus \mathbb{Q}_{\text{disqualified}}} \ \widehat{C}_{\text{ub}}(q^*) \leq \widehat{C}_{\text{lb}}(q_i).$$

If all plans are pruned but the halt condition is not triggered yet, PlanPruner-AC should additionally recover some plans that are not in $\mathbb{Q}_{\text{disqualified}}$, preferably those with the lowest cost.

However, a limitation of this approach is that accuracy estimation depends heavily on canary assumptions about their coverage. Future work could explore more effective algorithms, particularly those that adaptively update statistical estimates.

Table 2: Datasets, queries, and the number of available Clue instances for each query.

Query	Dataset	Length	Query Description	# Clues
D_1	WLD	414 min	leopard hunting event	4
D_2	WLD	414 min	leopards and monkeys	3
D_3	WLD	414 min	track of leopards	5
D_4	NBA	135 min	active play segments of James dunking	6
D_5	Traffic	350 min	cars turning left with people at night	5
D_6	Breakfast	162 min	baby making breakfast	4
D_7	VideoGame	78 min	number of game wins	3
D_8	Timelapse	43 sec	specific timestamp	1

5 EVALUATION

We now turn our attention to the evaluation of the algorithms proposed so far. We first detail the experimental setting, followed by our results.

5.1 Experimental Setup

Datasets, Queries and Clues. The real-world datasets employed in our evaluation are detailed below.

- WLD [12]: a wild life documentary dataset, comprising 15 films, complete with annotations for animal object detection and tracking, as well as subtitles.
- NBA: a recorded video of basketball games. Analysis of sports videos has been employed in prior work, such as [41, 42].
- Traffic: a surveillance video from a crossroad intersection. Traffic video analysis has been studied in prior work, such as [23, 28].
- Breakfast [32]: an action recognition dataset consisting of several cooking activities, such as preparation of coffee.
- VideoGame: live screen recordings of video game play[7].
- Timelapse: a YouTube time-lapse video[8] captured over extended periods, showcasing changes in city scenery over time.

Tb. 2 outlines the datasets, queries, and the number of available Clue instances used to optimize query processing[9]. For each query, we declared several Clue instances belonging to the five Clue types previously described. For example, consider the query D_1, which aims to identify all the video segments containing the event leopard hunting and employs an event detection model as the UDF to identify the events. For this query, we declare four available Clue instances: (1) C1, a PREFILTER type Clue instance that utilizes video captions to filter the frames by searching for keyword leopard; (2) C2, another PREFILTER type Clue instance that utilizes an object detection model to filter the frames by searching for object leopard; (3) C3, a BULK_SUBSTITUTE type Clue instance that offers a faster (but

[7]Available in the project repository.

[8]https://www.youtube.com/watch?v=JRH3rydhmKs

[9]Due to space constraints we cannot include the full specification of all queries and Clues we utilized in our evaluation. They are available in our technical report [9].

Table 3: The runtime of the original query plan of each query.

Query	Original Plan Runtime	Query	Original Plan Runtime
D_1	1130 min	D_5	411 min
D_2	376 min	D_6	486 min
D_3	562 min	D_7	12.2 min
D_4	913 min	D_8	78 sec

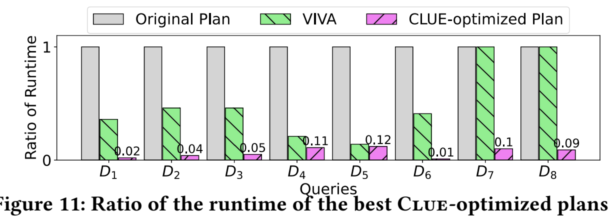


Figure 11: Ratio of the runtime of the best CLUE-optimized plans to the original runtime.

Table 4: Queries where individual CLUE instances of each type are most effective.

CLUE Types	Queries	CLUE Types	Queries
MONOTONIC	D_5,D_8	DISJOINT	D_3,D_4,D_7
DOMAIN_MAPPING	D_1,D_2,D_5	PREFILTER	D_1,D_2,D_4
BULK_SUBSTITUTE	D_1,D_2,D_6		

less accurate) object detection model; and (4) `C4`, a `DOMAIN_MAPPING` type CLUE instance that crops parts of each frame identified as sky, before forwarding them to the object detection model in `C2` and later adjusts the bounding box coordinates of the frame to its original size.

Metrics:

- *Runtime:* the time taken to execute a query plan. It serves as a measure to evaluate and compare the runtime efficiency of various query plans for a given query.
- *Optimization Time:* the time taken to select the best query plan from a set of candidates by evaluating them on the canary. It serves as a measure to assess and compare the optimization time required by various plan selection approaches.
- *Accuracy:* the query-wide accuracy of a query plan on the canary, compared against the ground truth outputs (or the outputs of the original query plan), as defined in Eq. 10.

Algorithms Compared. We generate an *original* query plan utilizing the parser and conventional query optimizer provided by a state-of-the-art VDBMS, EVA [49]. Other advanced optimization strategies proposed in EVA, such as UDF result reuse, are orthogonal to the proposal in this paper and are not included in our experiments. We compare the query runtime of the query plan optimized through our CLUE-based approach against the query plans produced by EVA and those produced by VIVA a state-of-the-art video analytics system [41].

- VIVA leverages relational hints about the relationships between machine learning models, which include two types: replace and filter. We generalize the definition of filters in VIVA to support other data sources (such as captions) and metadata (such as shooting time), beyond including additional models, to enable VIVA to utilize the `PREFILTER` or `BULK_SUBSTITUTE` CLUES that are identified for each query.

For the query optimization process (i.e. plan generation and selection), we compare our proposal with an approach we refer to as *PlanScan*. PlanScan operates by executing all candidate plans on the canary and then selecting the most efficient one.

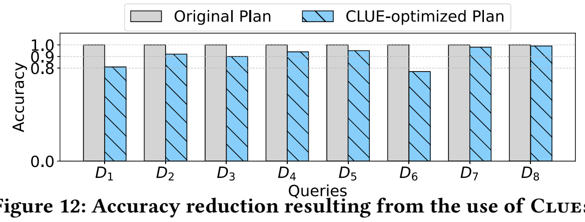


Figure 12: Accuracy reduction resulting from the use of CLUES.

5.2 Evaluation of CLUES

We first explore how CLUE instances enhance query processing, evaluated by *runtime*–the time taken to execute a query plan. Tb. 3 presents the runtime of executing the original query plan that is parsed and optimized by a conventional query optimizer in EVA [49] (as shown in Fig. 2), without applying any CLUES for each query enumerated in Tb. 2. Fig. 11 illustrates a comparison of the query runtime of the query plans optimized by `CSel` utilizing the CLUES, against both the runtime of the original plan (generated by EVA) and the plan optimized by VIVA. It presents the ratio of the optimized plan runtime over the original plan runtime. The comparison vividly demonstrates that the incorporation of CLUE instances leads to a significant reduction in query runtime. For example, for query D_1, the runtime required by the query plan optimized with CLUES is merely 0.02% of the original plan runtime; for D_6, this ratio drops to just 0.01%, which demonstrates the effectiveness of the CLUE-based query optimization system proposed in this paper.

Compared to VIVA, our CLUE-based optimization applies a broader range of optimization strategies, not limited to just model filter and model replace. As a result, it manages to reduce the query runtime *by 1-2 orders of magnitude on most queries* compared to VIVA. On some queries (such as D_7 and D_8), where suitable relational hints are absent or their effects are limited, VIVA is unable to optimize the query; in contrast, ours achieves a 10x speedup.

5.2.1 *Effectiveness of Individual CLUE Instances.*

In this section, we aim to gain insights into the most frequently effective CLUES per query. Tb. 4 displays the queries that individual CLUE instances of each type are most effective on. For detailed information on which CLUES were employed for each query and their effectiveness, refer to our technical report [9]. Here we illustrate the effectiveness of individual CLUE instances of each type through the following examples: the employment of a `MONOTONIC` type CLUE instance onto D_8 results in a reduction of its query runtime by approximately 90%; the `DISJOINT` type CLUE instance applied to D_7 leads to a 90% decrease in its runtime (this is the individual contribution of this filter out of the 3 applied to the reduction of the runtime); the `DOMAIN_MAPPING` type CLUE instance used in D_5 contributes a reduction in runtime by about 80%; the `PREFILTER` type CLUE instance for D_1 reduces its runtime by around 2 orders of magnitude; the employment of a `BULK_SUBSTITUTE` type CLUE instance to both D_1 and D_6 results in a decrease of their runtime by 50%.

5.2.2 *Combined Effect of CLUES.*

We investigate the combined impact of CLUES on query optimization through systematic experimentation. Specifically, for each query, we first conduct multiple experiments, optimizing it with a different single CLUE in each experiment, and observe the runtime reduction. Subsequently, we optimize each query using combinations of these single CLUES in each experiment. It allows us to compare the effects of individual versus combined CLUES on performance. If CLUES are unrelated,

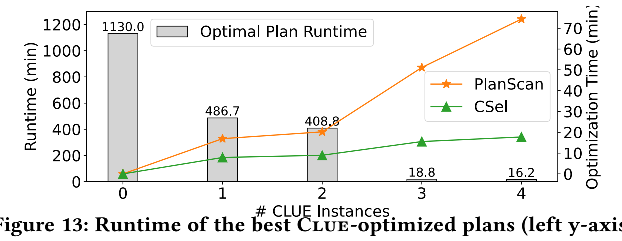


Figure 13: Runtime of the best CLUE-optimized plans (left y-axis) and the optimization time of different plan selection methods (right y-axis), varying the number of CLUE instances on query D_1.

such as the CLUES DISJOINT and PREFILTER applied to dataset D_4, then the combined effect nearly equals the product of their individual impacts. However, in most cases, the combined effect is less than the product of their individual impacts due to overlapping functionalities between the optimizations provided by the CLUES.

5.2.3 Accuracy of CLUE-Optimized Query Plans. Fig. 12 presents the accuracy of those CLUE-optimized query plans (the runtime of these plans was shown in Fig. 11). The query plan accuracy is defined as its query-wide accuracy (i.e. recall rate) when executed on the canary, compared against either the ground truth data or the outcomes produced by the original query plan, as specified in Eq. 10. It is evident that while our CLUES typically result in a modest accuracy reduction of no more than 10% for most queries, certain queries, such as D_1, might experience more significant accuracy impacts due to specific CLUES like BULK_SUBSTITUTE (which introduces a new machine learning model that brings additional uncertainty and errors). In these instances, VIVA imposes a similar accuracy reduction to that using CLUES. However it is imperative to have the flexibility to control accuracy and this underscores the need for the forthcoming evaluation of the CSel-AC algorithm, which incorporates accuracy constraints into plan selection, especially for these queries. In addition, we conducted experiments to assess whether the CLUES used affect precision, confirming that these CLUES do not impact it.

5.2.4 Impact of Varying CLUE Instance Number. Fig. 13 presents the runtime of the CLUE-optimized plans, varying the number of CLUE instances that are applicable (depicted using the bar chart on the left y-axis) for query D_1 so that we obtain versions of D_1 with varying number of CLUES. Using the notation introduced in §5.1 for D_1 we employ the CLUE instances as the following order: C2 (PREFILTER), C3 (BULK_SUBSTITUTE), C4 (DOMAIN_MAPPING), followed by C1 (PREFILTER). Hence, as the number of applied CLUE instances increases (from 0 to 4), the number of generated query plans also rises correspondingly (from 1 to 14), alongside a significant reduction in the query runtime of the best plan (from 1130 min to 16.2 min), showcasing a dramatic enhancement in query performance. This pattern is consistent across various queries; we use query D_1 as a representative example for brevity. This trend underscores the effectiveness of CLUE instances in optimizing overall runtime performance.

5.3 Evaluation of CSEL

We next evaluate the effectiveness of the plan selection approach proposed, CSel, by examining their *optimization time*–the time taken to select the best query plan from a set of candidates by evaluating them on the canary. Tb. 5 presents the optimization time for PlanScan over a variety of queries when applied to canary data sets. Fig. 14 provides a comparison of the optimization time required by the different plan selection approaches for each query, when applied to canary data sets. It presents the ratio of the optimization time over PlanScan. This evaluation highlights a key advantage of our proposed CSel method, which computes the cost bounds of plans and enables early pruning, avoiding exhaustive and precise estimation of runtime for the plans. As evident from the table, CSel demonstrates a substantial decrease in optimization time; for example, with query D_4, CSel reduces the optimization time by up to 95% in comparison to PlanScan.

Table 5: Optimization time of PlanScan on each query.

Query	PlanScan	Query	PlanScan	Query	PlanScan
D_1	74.3 min	D_4	12.1 min	D_7	1.0 min
D_2	5.7 min	D_5	113.6 min	D_8	28.8 sec
D_3	69.4 min	D_6	51.4 min		

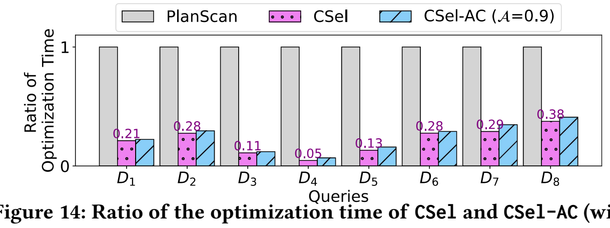


Figure 14: Ratio of the optimization time of CSel and CSel-AC (with a minimum tolerant accuracy $\mathcal{A}$=0.9) compared to PlanScan.

Fig. 13 presents the optimization time associated with CSel and PlanScan on its right y-axis, specifically in scenarios where the number of CLUE instances varies for query D_1. As expected, the increase in the number of CLUE instances results in a corresponding increase in the optimization time required for plan selection. With the increase in CLUE instances, which causes the search space of candidate plans to grow exponentially, the optimization time required by PlanScan similarly escalates in an exponential manner. However, the increase of optimization time for CSel is not proportional and is remarkably lower than that observed for PlanScan. When 2 CLUE instances are applied, the optimization time for CSel is roughly half of that required by PlanScan; when 4 CLUE instances are applied, the optimization time for CSel becomes only one-fifth of that required by PlanScan. This is due to the fact that the CSel algorithm, when executing a plan BLOCK, can update the bounds of all its relevant BLOCKS, thereby saving on optimization time. This pattern is consistent across various queries; we present results for query D_1 only for brevity. It indicates that CSel not only enhances the efficiency of query plan selection but does so in a manner that scales well with the complexity and number of CLUE instances involved.

Additionally, when a non-applicable CLUE is declared by the user, CSel can still determine the best query, but it slightly increases the optimization time. This is because a non-applicable CLUE increases the number of candidate plans generated by CGen, subsequently extending CSel's execution time. However, it is a minor penalty. According to our experiments, the increase in optimization time caused by a non-applicable CLUE typically amounts to less than $\frac{1}{n}$, where n is the total number of CLUES.

5.4 Evaluation of CSEL-AC

We turn our attention to the optimization problem *MinETOpt-AC*, where the objective is to identify the optimal plan with the least optimization time, while simultaneously ensuring that the

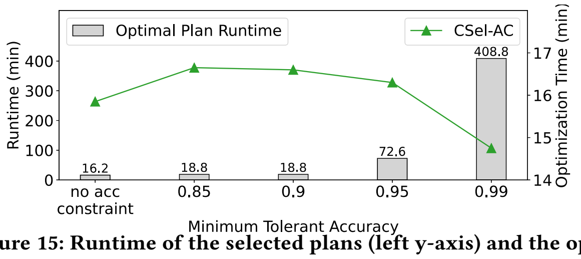


Figure 15: Runtime of the selected plans (left y-axis) and the optimization time of `CSel-AC` (right y-axis), varying the accuracy constraints (i.e. minimum tolerant accuracy) on query D_1.

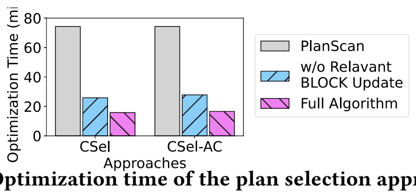


Figure 16: Optimization time of the plan selection approaches and their ablations on query D_1.

accuracy of the chosen plan exceeds a predefined minimum acceptable threshold. Fig. 14 presents a comparison of the optimization time of `CSel-AC` against both PlanScan and `CSel`. When compared with PlanScan, `CSel-AC` demonstrates a significant improvement, substantially reducing the optimization time. When compared to `CSel`, the optimization time of `CSel-AC` is slightly higher than that of `CSel`, in some queries (such as D_1). This increase in cost can be attributed to the potential need for executing a greater number of query plan Blocks to ensure that the plans meet the given accuracy constraint (i.e. minimum tolerant accuracy).

This figure highlights the ability of `CSel-AC` to strike a balance between efficiency and accuracy. Although in certain cases it marginally increases the optimization time compared to `CSel`, its importance becomes evident in situations where maintaining a high level of query accuracy is paramount.

5.4.1 Impact of Accuracy Constraints. Fig. 15 illustrates the optimization time of `CSel-AC` (right y-axis) and the runtime of the selected plans (left y-axis), while varying accuracy constraints for query D_1. As the minimum tolerant accuracy increases, some Clues can no longer be utilized, leading to an increase in the query runtime (from 16.2 min to 408.8 min). However, the optimization time of `CSel-AC` tends to be slightly higher compared to scenarios with no accuracy constraints or when the minimum tolerant accuracy are lower, due to the execution of additional query plan Blocks as explained earlier. Conversely, when the minimum tolerant accuracy becomes higher, many query plans that do not meet the constraint can be excluded early in the process. This early exclusion results in a lower optimization time for `CSel-AC` compared to the no accuracy constraint situation.

5.4.2 Ablation Study. Fig. 16 presents the optimization time for various plan selection approaches, including their ablated versions, applied to query D_1. For the purpose of comparison, we removed the mechanism for updating cardinality bounds based on Block relationships (see §4.3) from both `CSel` and `CSel-AC`, retaining only the bound pruning mechanism. This figure demonstrates that the optimization time for these ablated approaches is still significantly lower than that of PlanScan, but is greater than that of the full algorithm (i.e. `CSel` and `CSel-AC` without ablated), with an increase of about 60%-70%. This demonstrates that both the bound pruning mechanism and the mechanism for updating cardinality bounds based on Block relationships in our algorithms are effective and contribute to the efficiency improvement.

6 RELATED WORK

Video Database Management Systems. In the data management community, several recent works [1, 3–5, 8, 11, 13–16, 24, 26–28, 48] have introduced declarative query interfaces that utilize frame content (objects, spatial locations in the frame, etc), while optimizing for various parameters such as accuracy and/or execution. NoScope and BlazeIt [23, 25] utilize special-purpose-build neural networks (NNs) to detect objects accelerating queries via inference-optimized model search. Focus [21] implements low-latency search over large video datasets aiming to balance precision and query speed. SVQ [30, 46] provides a series of filters to accelerate video monitoring queries involving count and spatial constraints on objects present in the frames. [10, 47] present declarative query processing on video streams involving objects and their interactions. CORE [50] accelerates ML inference through predicate reordering, taking into account correlations in predicates. EVA [49] introduces a VDBMS that automatically materializes and reuses the results of costly UDFs to enhance faster exploratory data analysis. VIVA [41] presents a video analytics system that leverages relational hints to optimize SQL queries on video datasets. Our work extends this line of research, expanding the types of optimization strategies while proposing novel optimization algorithms demonstrating vast performance benefits to prior art.

Domain Knowledge Specification. The concept of specifying domain knowledge plays a crucial role in enhancing the performance and accuracy of both database systems and machine learning (ML) models [51]. Historically, the idea of feeding additional knowledge to a system to optimize query execution traces its origins to the early stages of query processing [29]. This approach is akin to the implementation of semantic integrity constraints in earlier computational models, as well as the utilization of hints in contemporary database systems like Microsoft SQL [36] and MySQL [37]. These hints serve as a guide to the database management system, suggesting more efficient ways to execute queries.

7 CONCLUSIONS

In this paper, we propose a novel framework for enhancing query optimization through domain-specific knowledge, introducing the concept of Clues. These Clues, diverse in types and associated with optimization rules, lay the groundwork for our proposed ClueVQS system, designed to automatically optimize queries for improving query processing efficiency. Further, we present algorithms, `CGen`, for query rewriting and plan generation, and `CSel`, for effective query plan selection, alongside its variant, `CSel-AC`, respecting user-specified accuracy constraints and allowing for trade-offs between query speed and query accuracy.

Future work could explore the extension of ClueVQS to support a wider array Clue types, and potentially employ machine learning techniques to enhance its capability for learning, summarizing, and autonomously utilizing the Clues, to further improve its adaptability and use experience.

REFERENCES

[1] Michael R Anderson, Michael Cafarella, German Ros, and Thomas F Wenisch. 2019. Physical representation-based predicate optimization for a visual analytics database. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1466–1477.

[2] Egon Balas and M Guignard. 1979. Report of the Session on: Branch and Bound-/Implicit Enumeration. In *Annals of Discrete Mathematics*. Vol. 5. Elsevier, 185–191.

[3] Jaeho Bang, Gaurav Tarlok Kakkar, Pramod Chunduri, Subrata Mitra, and Joy Arulraj. 2023. Seiden: Revisiting Query Processing in Video Database Systems. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2289–2301.

[4] Favyen Bastani, Songtao He, Arjun Balasingam, Karthik Gopalakrishnan, Mohammad Alizadeh, Hari Balakrishnan, Michael Cafarella, Tim Kraska, and Sam Madden. 2020. Miris: Fast object track queries in video. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1907–1921.

[5] Favyen Bastani and Samuel Madden. 2022. OTIF: Efficient Tracker Pre-processing over Large Video Datasets. In *Proceedings of the International Conference on Management of Data*. 2091–2104.

[6] Philipp Bergmann, Tim Meinhardt, and Laura Leal-Taixe. 2019. Tracking without bells and whistles. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 941–951.

[7] Alex Bewley, Zongyuan Ge, Lionel Ott, Fabio Ramos, and Ben Upcroft. 2016. Simple online and realtime tracking. In *2016 IEEE international conference on image processing (ICIP)*. IEEE, 3464–3468.

[8] Daren Chao, Yueting Chen, Nick Koudas, and Xiaohui Yu. 2023. Track Merging for Effective Video Query Processing. In *2023 IEEE 39th International Conference on Data Engineering*.

[9] Daren Chao, Yueting Chen, Nick Koudas, and Xiaohui Yu. 2024. Optimizing Video Queries with Declarative Clues (Technical Report). https://www.cs.toronto.edu/~drchao/papers/cluevqs_techreport.pdf.

[10] Daren Chao, Nick Koudas, and Ioannis Xarchakos. 2020. Svq++: Querying for object interactions in video streams. In *Proceedings of the International Conference on Management of Data*. 2769–2772.

[11] Daren Chao, Nick Koudas, and Xiaohui Yu. 2023. Marshalling Model Inference In Video Streams. In *2023 IEEE 39th International Conference on Data Engineering*.

[12] Kai Chen, Hang Song, Chen Change Loy, and Dahua Lin. 2017. Discover and learn new objects from documentaries. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 3087–3096.

[13] Yueting Chen, Xiaohui Yu, and Nick Koudas. 2020. TQVS: Temporal Queries over Video Streams in Action. In *Proceedings of the International Conference on Management of Data*. 2737–2740.

[14] Yueting Chen, Xiaohui Yu, and Nick Koudas. 2022. Ranked Window Query Retrieval over Video Repositories. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2776–2791.

[15] Yueting Chen, Xiaohui Yu, Nick Koudas, and Ziqiang Yu. 2021. Evaluating Temporal Queries Over Video Feeds. In *Proceedings of the International Conference on Management of Data*. 287–299.

[16] Pramod Chunduri, Jaeho Bang, Yao Lu, and Joy Arulraj. 2022. Zeus: Efficiently Localizing Actions in Videos using Reinforcement Learning. (2022).

[17] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.

[18] Databricks Inc. 2024. User-Defined Functions - Databricks Documentation. https://docs.databricks.com/en/udf/index.html.

[19] Beatrice Finance and Georges Gardarin. 1991. A rule-based query rewriter in an extensible dbms. In *Proceedings. Seventh International Conference on Data Engineering*. IEEE Computer Society, 248–249.

[20] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. 2017. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*. 2961–2969.

[21] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B Gibbons, and Onur Mutlu. 2018. Focus: Querying large video datasets with low latency and low cost. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 269–286.

[22] IBM. 2024. User Defined Functions in DB2. https://www.ibm.com/docs/en/db2/11.5?topic=functions-user-defined.

[23] Daniel Kang, Peter Bailis, and Matei Zaharia. 2019. BlazeIt: Optimizing Declarative Aggregation and Limit Queries for Neural Network-Based Video Analytics. *Proceedings of the VLDB Endowment* 13, 4 (2019), 533–546.

[24] Daniel Kang, Peter Bailis, and Matei Zaharia. 2019. Challenges and Opportunities in DNN-Based Video Analytics: A Demonstration of the BlazeIt Video Query Engine.. In *CIDR*.

[25] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. NoScope: Optimizing Deep CNN-Based Queries over Video Streams at Scale. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1586–1597.

[26] Daniel Kang, John Guibas, Peter Bailis, Tatsunori Hashimoto, Yi Sun, and Matei Zaharia. 2021. Accelerating approximate aggregation queries with expensive predicates. *arXiv preprint arXiv:2108.06313* (2021).

[27] Daniel Kang, John Guibas, Peter Bailis, Tatsunori Hashimoto, Yi Sun, and Matei Zaharia. 2024. Data Management for ML-based Analytics and Beyond. *ACM/JMS Journal of Data Science* 1, 1 (2024), 1–23.

[28] Daniel Kang, John Guibas, Peter Bailis, Tatsunori Hashimoto, and Matei Zaharia. 2022. TASTI: Semantic Indexes for Machine Learning-based Queries over Unstructured Data. (2022).

[29] Jonathan Jay King. 1979. *Exploring the use of domain knowledge for query processing efficiency.* Department of Computer Science, Stanford University.

[30] Nick Koudas, Raymond Li, and Ioannis Xarchakos. 2020. Video monitoring queries. In *IEEE International Conference on Data Engineering*. 1285–1296.

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2017. Imagenet classification with deep convolutional neural networks. *Commun. ACM* 60, 6 (2017), 84–90.

[32] Hilde Kuehne, Ali Arslan, and Thomas Serre. 2014. The language of actions: Recovering the syntax and semantics of goal-directed human activities. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 780–787.

[33] Michael A Laurenzano, Parker Hill, Mehrzad Samadi, Scott Mahlke, Jason Mars, and Lingjia Tang. 2016. Input responsiveness: using canary inputs to dynamically steer approximation. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 161–176.

[34] Yao Lu, Aakanksha Chowdhery, Srikanth Kandula, and Surajit Chaudhuri. 2018. Accelerating machine learning inference with probabilistic predicates. In *Proceedings of the 2018 International Conference on Management of Data*. 1493–1508.

[35] Mishaim Malik, Muhammad Kamran Malik, Khawar Mehmood, and Imran Makhdoom. 2021. Automatic speech recognition: a survey. *Multimedia Tools and Applications* 80 (2021), 9411–9457.

[36] Microsoft. 2024. MicrosoftSQL Hints (Transact-SQL). https://learn.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql-query?view=sql-server-ver16.

[37] MySQL. 2024. MySQL Optimizer Hints. https://dev.mysql.com/doc/refman/8.0/en/optimizer-hints.html.

[38] Karthik Ramachandra, Kwanghyun Park, K Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham. 2017. Froid: Optimization of imperative programs in a relational database. *Proceedings of the VLDB Endowment* 11, 4 (2017), 432–444.

[39] Joseph Redmon and Ali Farhadi. 2018. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767* (2018).

[40] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems* 28 (2015).

[41] Francisco Romero, Johann Hauswald, Aditi Partap, Daniel Kang, Matei Zaharia, and Christos Kozyrakis. 2022. Optimizing video analytics with declarative model relationships. *Proceedings of the VLDB Endowment* 16, 3 (2022), 447–460.

[42] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: A GPU cluster engine for accelerating DNN-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 322–337.

[43] Karen Simonyan and Andrew Zisserman. 2014. Two-stream convolutional networks for action recognition in videos. In *Advances in neural information processing systems*. 568–576.

[44] Snowflake Inc. 2024. User-Defined Functions Overview. https://docs.snowflake.com/en/developer-guide/udf/udf-overview.

[45] Zhaoguo Wang, Zhou Zhou, Yicun Yang, Haoran Ding, Gansen Hu, Ding Ding, Chuzhe Tang, Haibo Chen, and Jinyang Li. 2022. Wetune: Automatic discovery and verification of query rewrite rules. In *Proceedings of the 2022 International Conference on Management of Data*. 94–107.

[46] Ioannis Xarchakos and Nick Koudas. 2019. Svq: Streaming video queries. In *Proceedings of the International Conference on Management of Data*. 2013–2016.

[47] Yannis Xarchakos and Nick Koudas. 2021. Querying for interactions. In *IEEE International Conference on Data Engineering*. 2153–2158.

[48] Ran Xu, Jinkyu Koo, Rakesh Kumar, Peter Bai, Subrata Mitra, Sasa Misailovic, and Saurabh Bagchi. 2018. {VideoChef}: Efficient Approximation for Streaming Video Processing Pipelines. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 43–56.

[49] 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*. 602–616.

[50] Zhihui Yang, Zuozhi Wang, Yicong Huang, Yao Lu, Chen Li, and X Sean Wang. 2022. Optimizing machine learning inference queries with correlative proxy models. *Proceedings of the VLDB Endowment* 15, 10 (2022), 2032–2044.

[51] Lina Zhou, Shimei Pan, Jianwu Wang, and Athanasios V Vasilakos. 2017. Machine learning on big data: Opportunities and challenges. *Neurocomputing* 237 (2017), 350–361.