ABSTRACT

Title of Dissertation: ALGEBRAIC MULTIMEDIA: THEORY AND IMPLEMENTATION

Marat Fayzullin, Doctor of Philosophy, 2004

Dissertation directed by: Professor V. S. Subrahmanian
Computer Science

Storage, processing, and presentation of multimedia data, such as images, video, audio, or multimedia presentations, has become an important area of computer science. The goal of this dissertation is to formalize access methods to multimedia data by developing algebras that operate on PowerPoint presentations, video, and audio as in the case of relational algebra operating on tabular data. This dissertation also proposes ways to create summaries of multimedia data.

ALGEBRAIC MULTIMEDIA: THEORY AND IMPLEMENTATION

by

Marat Fayzullin

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland at College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2004

Advisory Committee:

Professor V. S. Subrahmanian, Chairman/Advisor
Professor Dana Nau,
Professor Lise Getoor,
Professor Stephen Kudla,
Professor Hanan Samet

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

During the last decade, the automated storage and processing of multimedia data, such as images, sound, video, and multimedia presentations, has become widely used in many aspects of human life. Whether you are preparing a speech for a business meeting, or searching the Internet for Christmas presents, or identifying a compact disk with Gracenote, it always comes down to computers having to work with media that is richer than the "traditional" numbers and text, i.e. with the *multimedia*.

Multimedia data processing can be roughly classified by purpose into (i) storage and delivery of data, (ii) indexing and search, (iii) data composition, and (iv) data summarization. Research in some of these areas has been more successful than others, and some work has even resulted in commercial tools (such as various multimedia search engines and authoring tools). In others, such as data summarization, the research still hasn't led to any widely commercialized tools.

Less work has been done on formalization of multimedia data models in the way relational algebra formalizes the tabular data model. Having formal models for multimedia data would help better integration between multimedia applications (i-iv) and allow for tools working across multiple media types.

Abstracting multimedia data is difficult mainly because of its variety. The data can be organized spatially (graphics), temporally (voice), or both (video). It may contain attributes as diverse as color distributions, motion vectors, frequency spectrums, phonemes, musical score, or plain text annotations. Even the same type of media (such as "graphics" or "video") can be represented in different ways with various file formats. Each format has its own merits and weaknesses, sometimes making conversion to other formats difficult or undesirable. For example, TrueColor images stored in bitmap files retain all of their pixel properties, but are often too large to archive, while more compact *JPEG* images lose individual pixel properties. Vector images (e.g. *SVG*) can be converted to bitmaps, but all the vector data will be lost in the process, making infeasible further geometrical transformations of the image. PowerPoint presentations, stored in the *PPT* format, can also be converted to sequences of vector or bitmap images, at the cost of losing their internal hierarchy of objects on slides.

Essentially, every multimedia file contains *partially structured data*. It is often useful to query these structures directly, as they get modified or lost during conversion to a more generic file format. The result of such a query will naturally appear in the same format as the input. For example, a query that selects all frames with a certain kind of motion from an MPEG source would produce an MPEG "summary" of that source.

In addition to the data contained in a file, there is sometimes other information that describes this data. This information may either come separately from the file, or be implied from the data in the file and stored for later usage. An MPEG video, for example, may have textual annotations that describe the video by the second. An audio file may be passed through a speech recognition program to produce a separate text transcript. Obtaining such information and using it in queries would be very

useful.

Finally, indexing can be applied both to the data stored in the file and the information that accompanies it. Finding the right index structures would facilitate query execution.

The purpose of this work is to devise algebras that abstract operations performed on some multimedia data, namely, PowerPoint presentations, video streams, and audio streams. The work generally involves proposing a formal model for the data, defining the algebra, finding and proving equivalences in the algebra, creating a reference implementation of the database engine and a query optimizer and, finally, evaluating the performance of this implementation.

The multimedia query languages developed in this work can be used as a part of bigger heterogeneous database and agent systems, such as TSIMMIS [14], HERMES [77], or IMPACT [68]. They can also be employed by the end users to create, search, transform, merge, split, and summarize multimedia documents.

For example, a university student may search a database of lecture presentations for slides containing a description of binary trees. He can issue a query to find these slides and combine them with slides on hashing to create a single personalized presentation.

Similarly, a business analyst may want to scan all PowerPoint presentations made by the department in the last year to find annual budget projections and attach the relevant slides to his own presentation comparing the projected and the actual budget use.

In a different example, a policeman looking at a surveillance video may request all fragments of this video where motion occurs and a certain object (such as a gun) is present. The result will be a video summary.

In yet another example of video summarization, a sports news editor can take an annotated video of a soccer play and create a video summary of all goals by one of the teams.

This thesis consists of seven chapters. Chapter 2 describes the pptA algebra that operates on PowerPoint presentations. This chapter (i) provides a formal model of a PPT presentation, (ii) defines basic pptA operators, both new and those similar to well known relational operators, (iii) shows algebraic equivalences, (iv) describes the pptA implementation and the cost model for this implementation, (v) describes the implementation of a query optimizer based on the cost model and shown equivalences, and (vi) covers experiments conducted to estimate the merits of the query optimization.

Chapter 3 covers the ADA algebra for audio recordings. The audio algebra model is based on the concept of streams that carry various representations of the same audio recording, such as waveforms, frequency spectrum, musical score, and text transcripts. Similarly to pptA , Chapter 3 (i) provides a formal model of audio files and databases, (ii) defines basic ADA operators, both new and similar to well known relational operators, (iii) shows algebraic equivalences, (iv) proposes and discusses several data structures facilitating the execution of some basic algebraic operators (v) describes a reference ADA implementation, and (vi) covers experiments conducted to assess the effect of data structures on the query execution.

Chapter 4 describes the VDA algebra for processing video data. The video algebra model is based on grouping video frames into blocks containing spatially localized features, such as certain colors, movement, objects, actions, or events. This model is closely related to the one used for video summarization in Chapter 5. Chapter 4 (i) defines the model formally, (ii) defines basic VDA operators, both new and

similar to well known relational operators, (iii) shows algebraic equivalences, (iv) discusses methods and data structures to accelerate operator execution and experimentally shows their benefits, (v) describes and experimentally verifies the cost model and the query optimizer, and finally (vi) describes the implementation of a reference VDA system.

Chapter 5 deals with video summarization, i.e. creation of shortened videos based on user defined constraints. The CPR summarization model proposed in this chapter rates summaries in terms of priority, continuity, and repetition. Chapter 5 (i) defines the concept of a video summary and its components, (ii) describes the language to specify constraints on a summary, (iii) shows ways to measure continuity, priority, and repetition of a summary, (iv) provides several algorithms to compute good summaries with respect to these three criteria, and (iv) describes experiments assessing the quality of summaries produced by different algorithms.

Chapter 6 presents a more general approach to document summarization using the same basic CPR model introduced in Chapter 5. The STORY system described in Chapter 6 allows to extract information about a given subject from multiple web documents, text documents, relational and XML data sources and create a text narrative (story) conveying this information to the user. Chapter 6 (i) defines basic concepts of entities, attributes, and stories, (ii) introduces generalization and conflict resolution in stories, (iii) provides ways to measure continuity, priority, and repetition of stories, (iv) discusses several algorithms to produce good stories with respect to these three criteria, and (v) describes our prototype STORY system implementation and the process for creating STORY applications.

The final Chapter 7 provides some concluding remarks.

Chapter 2

An Algebra For PowerPoint Presentations

2.1 Introduction

There are now millions of PowerPoint presentations on the web as well as on corporate intranets. There is a growing need to query large collections of such presentations. Corporate officials may want to find slides containing budget forecasts. Scientists may wish to find relevant slides of a colleague's technical presentations. University students studying Quattro may examine presentations on the web to get explanations that are more intuitive to them than their instructor's explanation. In short, the need to query such data sources cuts across a wide spectrum of end users.

Despite all this interest, there have been just two efforts (Ozsoyoglu et. al. [49, 46] and Adali et. al. [3]) to come up with formal models of general multimedia presentations — neither of these frameworks takes advantage of the specific features of PowerPoint. Both models query a presentation based on the *playout* of the presentation rather than the way the presentation is structured. Ozsoyoglu et. al. [49] use a graph representation of multimedia presentations and present a graph-based algebra. Adali et. al. [3] present a difference constraint based model and algebra for modeling

interactive presentations and querying them.

In contrast, PowerPoint presentations have a logical hierarchical representation consisting of three components — presentations, slides, and objects within a slide. In order to have an algebra that is specific to PowerPoint, one must take into account the specific representation of PowerPoint data which uses this logical structure. The work presented in this chapter [25] uses this hierarchy and thus captures PowerPoint presentations more accurately than previous models do. In contrast to [3], we have developed a prototype implementation of the PPT database and shown extensive equivalence results which may be used as rewrite rules for query optimization (which [46] does not). We have also developed a cost model for such databases (which to my knowledge is the first such cost model) and obtained experimental results assessing the use of these equivalences from an efficiency point of view (again a first to my knowledge).

The organization of this chapter is as follows. In the first section, I start by presenting a formal model of a PPT presentation. The second section describes the algebraic operators includes both analogs of traditional relational operators, and the new operators such as APPLY . This section also shows a host of query equivalence results that will be useful for optimizing queries. The implementation and the cost model are covered in the third section, together with the experimental validation of the cost model. The fourth section contains experimental results related to equivalences and the optimization. The final, fifth section compares the proposed framework with the existing body of work.

2.2 A Formal Description of PowerPoint Databases

A PPT presentation consists of a *sequence* of slides. Each slide contains a set of objects. Such objects can include text boxes, images, animations, embedded data views (e.g. charts and tables) amongst others. Each object has attributes — different objects may have different attributes and different attribute values. In order to formalize the definition of a PPT document, let us start "bottom up" by defining objects first, then slides, then presentations, and finally PPT databases.

Let us start with the assumption that there is some arbitrary but fixed universe $\mathcal{A}$ of strings called *attributes* and each attribute $A \in \mathcal{A}$ has an associated domain $dom(A)$. Assume the existence of a special set $\mathsf{Mand} \subseteq \mathcal{A}$ of attributes called *mandatory* attributes. Each domain $dom(A)$ may have zero or more associated binary relations in it. We will use the notation $BR(A)$ to denote the set of all binary relations associated with attribute A. For example, $dom(Type) = \mathbf{N}$ and $BR(Type) = \{=, \neq\}$. Similarly, if $dom(Data)$ is an arbitrary string, then $BR(Data) = \{=, \neq, <, \leq, >, \geq , contains\}$.

Note: Throughout this chapter, I assume that Mand contains at least the following attributes: $Type$, $Data$ (arbitrary strings), and Loc (two coordinate pairs representing top left and bottom right corners of a rectangle).

Definition 2.2.1 (Attribute Value Pair) *If $A \in \mathcal{A}$ is an attribute, and $v \in dom(A)$, then $\langle A, v \rangle$ is an* attribute value pair. *A is the* name *of this pair and v is the* value *of the pair. For example, $\langle Loc, (2, 3, 4, 5) \rangle$ is an attribute value pair.*

Example 2.2.1 *$Color$ and $Picture$ are possible attributes in $\mathcal{A}$. $dom(Color)$ may be an enumerated type consisting of strings that denote colors, and $dom(Picture)$ may be the set of strings denoting valid image names with extensions such as JPG,*

GIF, TIFF and BMP. Then $\langle Color, \text{“}red\text{”}\rangle$ *and* $\langle Picture, \text{“}aster.jpg\text{”}\rangle$ *are attribute value pairs. Another possible attribute is* $ColorSet$ *whose domain is the superset of* $dom(Color)$.

The following definition says that for a set of attribute value pairs to be considered valid, it must contain a pair for each mandatory attribute.

Definition 2.2.2 (Valid Attribute Value Pairs) *A finite set* AVP *of attribute value pairs is said to be* valid *iff for every attribute* $A \in$ Mand *there is exactly one pair in* AVP *of the form* $\langle A, -\rangle$. *We will use the* “ $-$ ” *symbol to denote “any value”.*

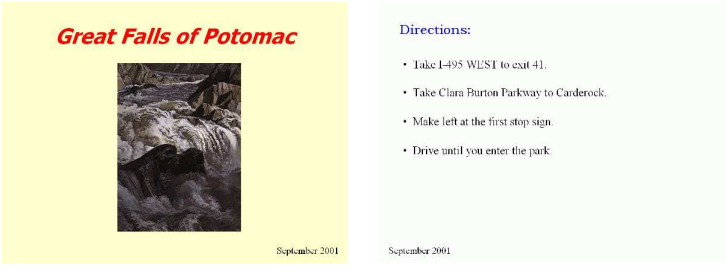
Example 2.2.2 $\{\langle Type, PICTURE\rangle, \langle Data, \text{“}aster.bmp\text{”}\rangle, \langle Loc, (5, 25, 55, 75)\rangle\}$ *is valid. However,* $\{\langle Data, \text{“}Directions\text{”}\rangle, \langle Color, \text{“}blue\text{”}\rangle, \langle Font, \text{“}Arial\text{”}\rangle\}$ *is not valid as it contains no pairs of the form* $\langle Type, -\rangle$ *and* $\langle Loc, -\rangle$.

Throughout the rest of this chapter, I assume the existence of a special set valued domain, $dom(\mathsf{IDs})$, i.e. every member of $dom(\mathsf{IDs})$ is a set.

Definition 2.2.3 (Object) *An object* o *is a pair* $o = \langle id, \mathsf{AVP}\rangle$ *where* id *is a member of* $dom(\mathsf{IDs})$, *and* AVP *is a valid set of attribute value pairs.*

Slide s_{11} Slide s_{12} Slide s_{13}

Figure 2.1: Presentation p_1.



Slide s_{21} Slide s_{22}

Figure 2.2: Presentation p_2.

Figures 2.1 and 2.2 show two very small PowerPoint presentations, "Falls.ppt" and "Flowers.ppt". We will use these presentations to illustrate the definitions and concepts introduced in this chapter.

Example 2.2.3 *For example, the first slide of the "Flowers" presentation has such objects as* $\langle\{o_{111}\}, \{\langle Type, TEXT\rangle, \langle Data, \text{“}Flowers\ of\ the\ Mountains\text{”}\rangle, \langle Loc, (20, 10, 620, 40)\rangle, \langle FontSize, 30\rangle, \langle Color, GREEN\rangle\}\rangle$, $\langle\{o_{112}\}, \langle Type, TEXT\rangle, \langle Data, \text{“}Presentation\ by\ John\ Smith\text{”}\rangle, \langle Loc, (220, 45, 420, 60)\rangle\}\rangle$, *and* $\langle\{o_{113}\}, \{\langle Type, PICTURE\rangle, \langle Data, \text{“}flowers.bmp\text{”}\rangle, \langle Loc, (20, 80, 320, 470)\rangle\}\rangle$.

Definition 2.2.4 (Slide) *A* slide *is a triple* $\langle id, Titles, Objects\rangle$ *where* $id \in dom(\mathsf{IDs})$, $Titles$ *is a finite set of strings, and* $Objects$ *is a finite set of objects such that if* $\langle o, \mathsf{AVP}_1\rangle, \langle o, \mathsf{AVP}_2\rangle \in Objects$ *then* $\mathsf{AVP}_1 = \mathsf{AVP}_2$.

The above definition requires that the same object id cannot occur twice in the set of objects comprising a slide. In addition, it is important to note that $Titles$ may be viewed as an attribute whose domain is the *powerset* of the set of all strings.

Example 2.2.4 *The first slide in the "Flowers.ppt" may be described as* $\langle\{s_{11}\}, \{\text{“}Title\ Slide\text{”}\}, \{objects\}\rangle$.

Definition 2.2.5 (PPT Presentation) *A* PPT presentation *is a* 4*-tuple* $(Titles, Authors, Slides, \mathsf{succ})$ *where:*

1. $Slides$ *is a finite set of slides such that:* $(\forall s_1, s_2 \in Slides)\, s_1 \neq s_2 \rightarrow s_1.id \neq s_2.id$
2. $Titles$ *and* $Authors$ *are finite sets of strings, and*
3. $\mathsf{succ} : Slides \rightarrow Slides$ *is an injective* partial *function that satisfies the following axioms:*
   (a) *there is exactly one member of* $Slides$*, denoted* $Slides_{end}$ *for which* $\mathsf{succ}(Slides_{end})$ *is undefined.*
   (b) $(\forall s_1, s_2 \in S)\mathsf{succ}(s_1) = \mathsf{succ}(s_2) \rightarrow s_1 = s_2$.

Note that as succ is injective, it implicitly induces a total ordering on the slides in a presentation. When $s_2 = \mathsf{succ}(s_1)$, we will often refer to s_2 as the successor of s_1 and s_1 as the predecessor of s_2. As in the case of slides, $Titles$ and $Authors$ may be viewed as attributes whose domain is the *powerset* of the set of all strings. Throughout the chapter, I will proceed with this assumption. Let us further assume that all set valued attributes A have set inclusion as a special binary relation in $BR(A)$. Furthermore, for any attribute A whose domain $dom(A)$ is a powerset type, instead of writing $\{x\} \subseteq X$, we will often write $x \in X$.

Definition 2.2.6 (PPT Database) *A* PPT database *is a finite set of* PPT *presentations.*

2.3 The PowerPoint Algebra pptA

In this section, I define the operators associated with the PowerPoint algebra, pptA . In addition to operators that resemble the classical relational algebra operators (selection,

projection, cartesian product, join, and set operators), pptA also has a number of operators that are unique to PowerPoint presentations. Many operators in pptA are parametrized by special functions that transform objects/slides/presentations into new objects/slides/presentations respectively.

Definition 2.3.1 (Transformation Functions) *An object (resp. slide, presentation) transformation function is a mapping from objects to objects (resp. slides to slides, presentations to presentations).*

The following examples present some sample transformation functions.

Example 2.3.1 *Function* $f_{red}(o) = \langle o.id, o.AVP - \{x \in o.AVP | x = \langle Color, - \rangle\} \cup \{\langle Color, \text{“}red\text{”}\rangle\}\rangle$ *changes the color of an object to “red”, while the* $f_{incfont}(o) = \langle o.id, o.AVP - \{x \in o.AVP \mid x = \langle Fsize, - \rangle\} \cup \{\langle Fsize, x + 4\rangle \mid \langle Fsize, x\rangle \in o.AVP\}\rangle$ *increases an object's font size by 4. Note that* $f_{incfont}$ *will not cause any change within an object that has no font properties.*

Example 2.3.2 *The slide transformation function* $f_{remsmall}(s) = \langle s.id, s.Titles, \{o \in s.Objects \mid \forall x \in o.AVP \;\; x.name = FontSize \rightarrow x.val \geq 12\}\rangle$ *removes all objects that have a font size of less than 12. Function* $f_{grback}(s) = \langle s.id, s.Titles, (s.Objects - \{x \mid x.name = Background\}) \cup \{\langle Background, green\rangle\}\rangle$ *changes backgrounds of all slide objects to green.*

Example 2.3.3 *An example of a presentation transformation function is one that adds a new author* a *to a presentation:*

$$f_{addauth}(p) = \langle p.Titles, p.Authors \cup \{\mathsf{a}\}, p.Slides, p.succ\rangle.$$

Throughout the rest of this chapter, I will often use $f(X)$ to denote the application of a transformation function to X. By looking at the type of X, one can infer whether f is an object (resp. slide, presentation) transformation function. Thu it is not necessary to explicitly state what kind of transformation function f is.

Definition 2.3.2 (A-invariant Transformation Function) *An object transformation function f is said to be A-invariant if for all objects o, $\langle A, v\rangle \in o.AVP$ iff $\langle A, v\rangle \in f(o).AVP$.*

Definition 2.3.3 (Commuting Transformation Functions) *Two object transformation functions f, g* commute *iff for all objects o, $f(g(o)) = g(f(o))$. Commuting slide and presentation transformation functions may be defined analogously.*

Definition 2.3.4 (Idempotent Transformation Functions) *A transformation function f is* idempotent *iff $f(X) = f(f(X))$.*

Definition 2.3.5 (Sensible Object Transformation Functions) *An object transformation function f is* sensible *iff for each attribute A, there is a bijection $f_A : dom(A) \rightarrow dom(A)$ such that for all objects o, $\langle A, v\rangle \in f(o).\mathsf{AVP}$ iff there exists an $\langle A, v'\rangle \in o.\mathsf{AVP}$ and $f_A(v) = v'$.*

The definition of sensible transformations may be extended to slides and presentations in the obvious way. Object transformation functions map objects to objects. Consider two objects o_1, o_2 with attribute value pairs $\{\langle A_1, v_1\rangle, \langle A_1, v_2\rangle\}$ and $\{\langle A_1, v_1\rangle, \langle A_2, v_2\rangle\}$. These two objects are indistinguishable (they have the same attributes and the same values for those attributes). However, theoretically, an object transformation function could map these two objects to two completely different objects (with varying attributes and different attribute values). The definition of sensibility says that this cannot happen - if two objects have the same pair $\langle A, v\rangle$ in their

set of attribute value pairs, the output (after the object transformation) must change this pair in the same way for both objects.

Throughout the rest of this chapter, unless specified otherwise, all transformation functions are assumed to be sensible. The example transformation functions above are all sensible.

2.3.1 The APPLY Operator

The first algebraic operator is the APPLY operator α which applies a transformation function to presentation entities.

Definition 2.3.6 (APPLY Operator) *Suppose f is a transformation function. The* APPLY *operator α is defined as follows.*

- *Suppose f is an object transformation function. Then the application of f to a an object o (resp. slide s, presentation p, presentation database pDB) is defined as:*

$$\begin{aligned}
\alpha(f,o).id &= o.id \\
\alpha(f,o).AVP &= f(o).AVP \\
\alpha(f,s).id &= s.id \\
\alpha(f,s).Titles &= s.Titles \\
\alpha(f,s).Objects &= \{\alpha(f,o) \mid o \in s.Objects\} \\
\alpha(f,p).Titles &= p.Titles \\
\alpha(f,p).Authors &= p.Authors \\
\alpha(f,p).Slides &= \{\alpha(f,s) \mid s \in p.Slides\} \\
\alpha(f,p).\mathsf{succ}(\alpha(f,s)) &= \alpha(f,s') \textit{ iff } p.succ(s) = s'
\end{aligned}$$

$$\alpha(f, pDB) = \{\alpha(f, p) \mid p \in pDB\}$$

- *Suppose f is a slide transformation function. Then the application of f to a slide s (resp. presentation p, presentation database pDB) is defined as:*

$$\begin{aligned}
\alpha(f, s).id &= s.id \\
\alpha(f, s).Titles &= f(s).Titles \\
\alpha(f, s).Objects &= f(s).Objects \\
\alpha(f, p).Titles &= p.Titles \\
\alpha(f, p).Authors &= p.Authors \\
\alpha(f, p).Slides &= \{\alpha(f, s) \mid s \in p.Slides\} \\
\alpha(f, p).\mathsf{succ}(\alpha(f, s)) &= \alpha(f, s') \textit{ iff } p.succ(s) = s' \\
\alpha(f, pDB) &= \{\alpha(f, p) \mid p \in pDB\}
\end{aligned}$$

- *Suppose f is a presentation transformation function. Then the application of f to a presentation p (resp. presentation database pDB is defined as:*

$$\begin{aligned}
\alpha(f, p) &= f(p) \\
\alpha(f, pDB) &= \{\alpha(f, p) \mid p \in pDB\}
\end{aligned}$$

Example 2.3.4 *$\alpha(f_{red}, s_{11})$ highlights all objects on slide s_{11} of "Flowers.ppt" in red.*

The following theorem states that APPLY commutes as long as the transformation functions commute and are sensible.

Theorem 2.3.1 *Suppose f, g are sensible object (resp. slide, presentation) transformation functions and suppose $X \in \{o, s, p\}$ where o is an object, s is a slide, p is a presentation. If f, g commute, then so does the* APPLY *operator, i.e.* $\alpha(f, \alpha(g, X)) = \alpha(g, \alpha(f, X))$.

Proof of Theorem 2.3.1. We show the case where f is an object transformation function (the cases when its a slide/presentation transformation function are similar). We need to show that each of the components of the object returned by $\alpha(f, \alpha(g, X))$ and $\alpha(g, \alpha(f, X))$ are identical where X is an object.

1. $\alpha(f, \alpha(g, X)).id = X.id = \alpha(g, \alpha(f, X)).id$.

2. Suppose $\langle A, v\rangle \in \alpha(f, \alpha(g, X)).AVP$. Then $\langle A, v\rangle \in f(g(X)).AVP$ As f, g commute, $\langle A, v\rangle \in g(f(X)).AVP$ and hence $\langle A, v\rangle \in \alpha(g, \alpha(f, X)).AVP$. The reverse inclusion is similar.

The following theorem states that the APPLY operator is idempotent as long as the transformation functions are themselves idempotent.

Theorem 2.3.2 *Suppose f is an idempotent object (resp. slide, presentation) transformation functions and suppose $X \in \{o, s, p\}$. Then $\alpha(f, \alpha(f, X)) = \alpha(f, X)$.*

Proof of Theorem 2.3.2. We show the case where f is an object transformation function (the cases when its a slide/presentation transformation function are similar). We need to show that each of the components of the object returned by $\alpha(f, \alpha(f, X)) = \alpha(f, X)$ coincide.

1. $\alpha(f, \alpha(f, X)).id = X.id = \alpha(f, X).id$

2. Suppose $\langle A, v\rangle \in \alpha(f, \alpha(f, X))$. Then $\langle A, v\rangle \in f(f(X)).AVP$ - as f is idempotent, $\langle A, v\rangle \in f(X).AVP$ which means it is also in $\alpha(f, X).AVP$. The reverse inclusion is similar.

2.3.2 The SELECT Operator

The most important operation is selection. Let us first define the syntax of a selection condition, and then define selection.

Selection Conditions

Let us assume the existence of special disjoint sets V_o, V_s, V_p whose members are called *root* object/slide/presentation variables, respectively.

Definition 2.3.7 (Object Variables) *(i) Every root object variable is an object variable of type "all." (ii) If X is an object variable and $A \in \mathcal{A}$ is an attribute, then $X.A$ is an object variable of type $dom(A)$* [1].

Slide variables and presentation variables may be defined in exactly the same way as object variables, by replacing all occurrences of the word "object" in the above definition by "slide" and "presentation" respectively. Hence, if S is a root slide variable, then $S.Titles$ is a perfectly good slide variable.

Definition 2.3.8 (Object Term) *(i) Every member of $dom(A)$ is an* object term *of type $dom(A)$. (ii) Every object variable of type τ is an* object term *of type τ. Slide terms and presentation terms are defined in an analogous way.*

[1] Assume that all such variables are well-typed.

Definition 2.3.9 (Object Atom) *If T_1, T_2 are object terms of type $dom(A)$ for some A, and if $\mathsf{op} \in BR(A)$, then T_1 op T_2 is an object atom. Slide atoms and presentation atoms are defined in an analogous way.*

Example 2.3.5 *If $dom(FontSize) = \mathbf{N}$ and "$\leq$"$\in BR(FontSize)$, then $o_1.FontSize \leq 24$ is an object atom. If $dom(Data)$ is the set of all strings and "contains"is in $BR(Data)$, then $contains(o_1.Data, o_2.Data)$ is an object atom.*

Definition 2.3.10 (Object Condition (OC) and Object Selection Condition (OSC)) *(i) Every object atom is an OC. (ii) If oc_1, oc_2 are OCs, then so are $oc_1 \wedge oc_2$, $oc_1 \vee oc_2$ and $\neg oc_1$. An* object selection condition *(OSC) is a special kind of OC that contains exactly one root variable, denoted $O \in V_o$.*

Slide selection conditions (SSCs) and presentation selection conditions (PSCs) are defined in an analogous way - the only difference is that the root variable is required to be in V_s and V_p respectively.

Example 2.3.6 *$\neg contains(O_1.Data, O_2.Data)$ and $O_1.FontSize \leq O_2.FontSize \wedge O_1.Data = O_2.Data$ are both object conditions. However, neither of them is an OSC because they contain two root variables. $O.FontSize < 20 \wedge contains(O.Data, \text{"}the\text{"})$ is a valid OSC as it has exactly one root variable O.*

Selection

We are now ready to define the selection operator. Given a selection condition C involving a free variable X, let us use $C[X/x]$ to denote the replacement of all occurrences of X in C by x.

Definition 2.3.11 (SELECT Operator) *Suppose C is a selection condition (OSC, SSC, or PSC) with the single root variable (either O ranging over objects or S ranging over slides) in it. The* SELECT *operator σ is defined as follows.*

- *Suppose C is an OSC. Then:*

$$\begin{aligned}
\sigma_C(s).id &= s.id \\
\sigma_C(s).Titles &= s.Titles \\
\sigma_C(s).Objects &= \{o \mid o \in s.Objects \wedge C[O/o]\} \\
\sigma_C(p).Titles &= p.Titles \\
\sigma_C(p).Authors &= p.Authors \\
\sigma_C(p).Slides &= \{\sigma_C(s) \mid s \in p.Slides\} \\
\sigma_C(p).succ &= \mathsf{succ}' \textit{ as defined below} \\
\sigma_C(pDB) &= \{\sigma_C(p) \mid p \in pDB\}
\end{aligned}$$

- *Suppose C is an SSC. Then:*

$$\begin{aligned}
\sigma_C(p).Titles &= p.Titles \\
\sigma_C(p).Authors &= p.Authors \\
\sigma_C(p).Slides &= \{\sigma_C(s) \mid s \in p \wedge C[S/s]\} \\
\sigma_C(p).\mathsf{succ} &= \mathsf{succ}' \textit{ as defined below} \\
\sigma_C(pDB) &= \{\sigma_C(p) \mid p \in pDB\}
\end{aligned}$$

- *Suppose C is a PSC. Then:*

$$\sigma_C(pDB) = \{\sigma_C(p) \mid p \in pDB \wedge C[P/p]\}$$

In the above, $\mathsf{succ}'(s_1) = s_2$ *where* $s_2 = \min\{s' \mid (\exists i > 0)(s' = \mathsf{succ}^i(s_1) \wedge s' \in \sigma_C(p).Slides) \wedge (\forall 0 < j < i)(\mathsf{succ}^j(s_1) \notin \sigma_C(p).Slides)\}$.

Example 2.3.7 *The user can select all pictures from slide* s_{13} *in the sample presentation* p_1 *("Flowers") by writing* $\sigma_{O.Type=PICTURE}(s_{13})$*. He can select all pictures in the presentation by writing* $\sigma_{O.Type=PICTURE}(p_1)$*.*

Just like in relational algebra, it is permissible to swap two SELECT operators in pptA :

Theorem 2.3.3 (Changing Order of SELECT Operators) *Suppose* C_1, C_2 *are selection conditions and* X *is a slide (resp. presentation). Then* $\sigma_{C_2}(\sigma_{C_1}(X)) = \sigma_{C_1}(\sigma_{C_2}(X))$*.*

Proof of Theorem 2.3.3. Similar to the proof of commutativity of a classical relational selection.

The following result shows that as long as an object (resp. slide) transformation is A-invariant for all A occurring in a selection condition, the APPLY operator can be pulled back outside a selection[2].

Theorem 2.3.4 (Pullback Theorem: APPLY and SELECT) *Suppose* f *is an object (resp. slide) transformation function that is* A*-invariant for all attribute names* A *occurring in an object (resp. slide) selection condition* C*, and* X *is a slide (resp. presentation). Then* $\alpha(f, \sigma_C(X)) = \sigma_C(\alpha(f, X))$*.*

Proof of Theorem 2.3.4. We show the case where f is an object transformation function (the cases when its a slide/presentation transformation function are similar).

[2]In classical relational algebra, we often like to *push* a selection inside a join. It will turn out that *pulling back* an apply out of various operators is often computationally efficient.

We need to show that each of the components of the object returned by $\alpha(f, \sigma_C(X))$ and $\sigma_C(\alpha(f, X))$ coincide.

1. $\alpha(f, \sigma_C(X)).id = X.id = \sigma_C(\alpha(f, X)).$

2. Suppose $o \in \alpha(f, \sigma_C(X)).Objects$. Then $o = f(x)$ for some x that satisfies condition C. As f is A-invariant for all attributes occurring in C, this means that $f(x)$ satisfies condition C as well. This in turn means that $o = f(x)$ is returned by $\sigma_C(\alpha(f, X))$. The reverse inclusion is similar.

3. $\alpha(f, \sigma_C(X)).Titles = X.Titles = \sigma_C(\alpha(f, X)).Titles.$

Basically, the theorem says that as long as the selection condition only evaluates attributes not affected by an APPLY operation, the order in which the two are performed is irrelevant.

Finally, to select slides containing a certain object inside a presentation, let us introduce the *slide select operator*.

Definition 2.3.12 (Slide SELECT Operator) *Given a presentation p and an object selection condition C, the slide select operator $\zeta_C(p)$ produces a new presentation such that*

$$\begin{aligned}
\zeta_C(p).Titles &= p.Titles \\
\zeta_C(p).Authors &= p.Authors \\
\zeta_C(p).Slides &= \{s \mid s \in p.Slides \wedge (\exists o \in s.Objects)\ C[O/o]\ is\ true\} \\
\zeta_C(pDB) &= \{\zeta_C(p) \mid p \in pDB\}
\end{aligned}$$

2.3.3 The PROJECT Operator

In this section, I define the pptA analog of projection. Not all attributes may be projected out. For instance, the object location attribute Loc cannot be projected out as PowerPoint requires each object to occupy some location on a slide.

Definition 2.3.13 (Projectable Attribute Set) *A* projectable attribute set *is any set* $\mathcal{A}'$ *of attributes such that* $\{Titles, Authors\} \cup \mathsf{Mand} \subseteq \mathcal{A}' \subseteq \mathcal{A}$.

Intuitively, projectable attribute sets prevent users from eliminating mandatory attributes and/or specialized attributes like $Titles, Authors$.

Definition 2.3.14 (Projection) *Suppose* $\mathcal{A}'$ *is a projectable attribute set. The* PROJECT *operator* $\pi_{\mathcal{A}'}()$ *is defined as follows.*

1. *If* o *is an object, then* $\pi_{\mathcal{A}'}(o) = \langle o.id, \{\langle A, v\rangle \mid \langle A, v\rangle \in o.\mathsf{AVP} \wedge A \in \mathcal{A}'\}\rangle$.
2. *If* s *is a slide, then* $\pi_{\mathcal{A}'}(s) = \langle s.id, s.Titles, \{\pi_{\mathcal{A}'}(o) \mid o \in s.Objects\}\rangle$.
3. *If* p *is a presentation, then* $\pi_{\mathcal{A}'}(p) = \langle p.Titles,\ p.Authors,\ \{\pi_{\mathcal{A}'}(s) \mid s \in p.Slides\}, p.\mathsf{succ}\rangle$.

Example 2.3.8 *Suppose the user wants to convert all slides in the sample presentation* p_2 *("Falls") to black and white. To do so, she may execute the projection operation* $\pi_{\overline{\{Color\}}}(p_2)$ *to remove all* $Color$ *attributes from all objects in all slides in* p_2. *Note that as usual* $\overline{\{Color\}}$ *denotes the complement of the set* $\{Color\}$ *and hence this operation only eliminates the color attribute of an object. Such a query may therefore be used to view a presentation in black and white.*

The following theorem tells us that the operations of APPLY and PROJECT commute.

Theorem 2.3.5 (Pullback Theorem: APPLY **and** PROJECT **)** *Suppose f is a sensible transformation function and $X \in \{o, s, p\}$. Then $\alpha(f, \pi_{\mathcal{A}'}(X)) = \pi_{\mathcal{A}'}(\alpha(f, X))$.*

Proof of Theorem 2.3.5. We show the case where f is an object transformation function (the cases when its a slide/presentation transformation function are similar). We need to show that each of the components of the object returned by $\alpha(f, \pi_{\mathcal{A}'}(X))$ and $\pi_{\mathcal{A}'}(\alpha(f, X))$ coincide. We consider the case where X is a slide (the case when presentations are considered is similar).

1. $\pi_{\mathcal{A}'}(\sigma_C(X)).id = X.id = \sigma_C(\pi_{\mathcal{A}'}(X)).id.$

2. $\pi_{\mathcal{A}'}(\sigma_C(X)).Titles = X.Titles = \sigma_C(\pi_{\mathcal{A}'}(X)).Titles.$

3. Suppose $\alpha(f, \pi_{\mathcal{A}'}(X)).Objects$. Then $o = f(o')$ where $o' = \pi_{\mathcal{A}'}(o'')$ for some $o'' \in X$. As f is sensible, if $\langle A, v'' \rangle \in o''.AVP$, then there is an $\langle A, f(v'') \rangle \in o.AVP$. It follows immediately that $\langle A, f(v'') \rangle \in o \in \pi_{\mathcal{A}'}(\alpha(f, X))$. The reverse inclusion is similar.

The above theorem says that one can execute APPLY and PROJECT operations in any order. The requirement of sensibility of f in the above theorem is important - without it, the theorem would not hold. The reason is that when we do a PROJECT , some attributes of objects might be eliminated. The sensibility requirement ensures that the transformation function behaves the same way on attributes common to an object and the projection of that object on the fields in $\mathcal{A}'$. The following theorem says that PROJECT and SELECT commute as long as all attributes mentioned in the selection condition also occur in the set of attributes involved in the projection.

Theorem 2.3.6 *Suppose C is a selection condition all of whose attributes occur in a projectable attribute set $\mathcal{A}'$. Then it is true that $\pi_{\mathcal{A}'}(\sigma_C(X)) = \sigma_C(\pi_{\mathcal{A}'}(X))$.*

Proof of Theorem 2.3.6. The proof follows immediately from the fact that neither selection nor projection change the values of any attributes.

2.3.4 The RENAME Operator

A common operator in relational databases is the renaming operator that provides new names to attributes.

Definition 2.3.15 (Renaming Function) *Suppose $\mathcal{A}, \mathcal{B}$ are disjoint sets of attribute names. A* renaming function *r is an injective mapping from $\mathcal{A}$ to $\mathcal{B}$.*

We will often use notation such as $\{A_1 \rightarrow B_1, A_2 \rightarrow B_2\}$ to indicate that a renaming function renames attribute A_1 by B_1 and A_2 by B_2.

Definition 2.3.16 (RENAME Operator) *Suppose r is a renaming function from $\mathcal{A}$ to $\mathcal{B}$. The* renaming operator *is defined as follows.*

- *The renaming of an object o, denoted $\rho^r(o)$ is the object $\langle o.id, \{\langle r(A), v\rangle \mid A \in \mathcal{A}\}\rangle$.*
- *The renaming of a slide s, denoted $\rho^r(s)$ is the slide $\langle s.id, s.Titles, s.Authors, \{\rho^r(o) \mid o \in Objects\}\rangle$.*
- *The renaming of a presentation p, denoted $\rho^r(p)$ is the presentation $\langle p.Titles, p.Authors, \{\rho^r(s) \mid s \in p.Slides\}, \mathsf{succ}'\rangle$ such that $\mathsf{succ}'(\rho^r(s)) = \rho^r(s')$ iff $\mathsf{succ}(s) = s'$.*
- *The renaming of a presentation database pDB, denoted $\rho^r(DB)$ is the presentation database $\{\rho^r(p) \mid p \in pDB\}$.*

2.3.5 The Cartesian Product

In this section, I define the Cartesian Product operation which forms the basis for JOIN .

Definition 2.3.17 (Attribute Merge Function) *An* attribute merge function *is a mapping g that takes as input an attribute name A and a set of values from $dom(A)$, and returns as output, a value in $dom(A)$. When the second argument of g is a singleton $\{v\}$, then $g(A, \{v\}) = v$.*

Attribute merge functions are used in the Cartesian Product operation to merge attributes that have the same name. In classical relational algebra this situation is avoided by insisting on renaming attributes with the same name, but this is perhaps more restrictive than we need here.

Example 2.3.9 *When merging object locations, we may want to return as output, a bounding box that bounds the rectangles associated with the two input locations. This is an attribute merge function $g_{mbr}(Loc, \{(lx_1, ly_1, rx_1, ry_1), (lx_2, ly_2, rx_2, ry_2)\}) = (min(lx_1, lx_2), min(ly_1, ly_2), max(rx_1, rx_2), max(ry_1, ry_2))$. Likewise, when merging object data strings, one may want to use a simple concatenation with carriage returns (CRs) between the strings. This can be defined as the attribute merge function $g_{cat}(Data, \{s_1, \ldots, s_n\}) = s_1\ CR\ \ldots\ CR\ s_n$ where CR is the carriage return character.*

Definition 2.3.18 (Object Product) *Suppose o_1, o_2 are objects and g is an attribute merge function. The* Cartesian Product *of o_1, o_2 under attribute merge function g, denoted $o_1 \times^g o_2$ is the object o such that:*

1. $o.id = o_1.id \cup o_2.id$;

2. $o.\mathsf{AVP} = \{\langle A, v\rangle \mid (\langle A, v\rangle \in o_1.\mathsf{AVP}$ *and there is no attribute-value pair of the form* $\langle A, -\rangle \in o_2.\mathsf{AVP})$ *or* $(\langle A, v\rangle \in o_2.\mathsf{AVP}$ *and there is no attribute-value pair of the form* $\langle A, -\rangle \in o_1.\mathsf{AVP})$ *or there exists* $\langle A, v_1\rangle \in o_1.\mathsf{AVP}$ *and* $\langle A, v_2\rangle \in o_2.\mathsf{AVP}$ *and* $g(A, \{v_1, v_2\}) = v\}$.

Example 2.3.10 *Suppose we merge the title* (o_{111}) *and the subtitle* (o_{112}) *objects on slide* s_{11} *("Flowers.ppt") using* g_{cat} *and* g_{mbr}. *The resulting object* $o_{111} \times^{g_{cat}, g_{mbr}} o_{112} = \langle\{o_{111}, o_{112}\}, \{\langle Type, TEXT\rangle, \langle Data, x\rangle, \langle Font, Helvetica\rangle, \langle FontSize, 26\rangle, \langle Loc, (20, 10, 620, 60)\rangle\}\rangle$ *where* x *is the string "Flowers of the Mountains CR Presentation by John Smith".*

Definition 2.3.19 (Slide Product) *The* CPRODUCT *of two slides* s_1, s_2 *under the attribute merge function* g, *denoted* $s_1 \times^g s_2$, *is the slide* s *such that:*

1. $s.id = s_1.id \cup s_2.id$.
2. $s.Titles = s_1.Titles \cup s_2.Titles$.
3. $s.Objects = \{o_1 \times^g o_2 \mid o_1 \in s_1.Objects \wedge o_2 \in s_2.Objects\}$.

For example, when two slides s_1, s_2 have two attributes A_1, A_2 in common, then the single slide s in the slide product assigns to A_1, the value $g(A_1, \{v_1, v_1'\})$ and $g(A_2, \{v_2, v_2'\})$ where v_1 (resp. v_2) is the value of attribute A_1 in s_1 and v_1' (resp. v_2') is the value of attribute A_2 in s_2. An example of CPRODUCT of two slides is given below.

Example 2.3.11 *The Cartesian product of slides* s_{11} *and* s_{12} *("Flowers" presentation) yields the slide given by* $s_{11} \times^g s_{12} = \langle\{s_{11}, s_{12}\}, \{$"$Title\ Slide$", "$Landscape\ Photos$"$\}, \{object\ cross\ products\}\rangle$.

The CPRODUCT of two presentations p_1, p_2 is defined in exactly the same way as slide product except that all occurrences of s_1, s_2 are replaced by p_1, p_2 respectively and all occurrences of o_1, o_2 are replaced by s_1, s_2 respectively. In addition, when defining CPRODUCT of two presentations, we need to specify the succ function. Consider a slide $s = s_1 \times^g s_2 \in p_1 \times^g p_2.Slides$. If $\mathsf{succ}(s_2)$ exists and equals s_2', then $\mathsf{succ}(s) = s_1 \times^g s_2'$. If $\mathsf{succ}(s_2)$ does not exist, then $\mathsf{succ}(s) = \mathsf{succ}(s_1) \times^g p_2.Slides_{start}$ where $\mathsf{succ}(s_1)$ denotes the successor of s_1 according to presentation p_1. If $\mathsf{succ}(s_1)$ does not exist, then $\mathsf{succ}(s) = s_1 \times^g \mathsf{succ}(s_2)$ assuming $\mathsf{succ}(s_2)$ exists. If neither $\mathsf{succ}(s_1), \mathsf{succ}(s_2)$ exist, then s is the last slide in the product.

The relationship between different pptA operators and CPRODUCT is complex because of attribute merge functions. In general, Cartesian Product is not commutative as attribute merge functions may not commute.

Definition 2.3.20 (Compatible Functions) *Suppose f is an object transformation function and g is an attribute merge function. f and g are said to be* compatible *iff for all objects o, all attributes A, and all sets V of values from $dom(A)$, it is the case that $g(A, \{f(v) \mid v \in V\}) = v'$. where $f(o).\mathsf{AVP}$ contains the pair $\langle A, v' \rangle$ and no other pair of the form $\langle A, - \rangle$ is in $f(o).\mathsf{AVP}$.*

Theorem 2.3.7 (Pullback Theorem: APPLY and CPRODUCT) *Suppose f is a sensible object transformation function and g is an attribute merge function such that f, g are compatible. Suppose X_1 and X_2 are both objects (or both slides, or both presentations). Then $\alpha(f, X_1 \times^g X_2) = \alpha(f, X_1) \times^g \alpha(f, X_2)$.*

Proof of Theorem 2.3.7. We show the theorem when X_1, X_2 are both objects. The cases when they are both slides/presentations are similar.

1. $\alpha(f, X_1 \times^g X_2).id = X_1.id \cup X_2.id = \alpha(f, X_1) \times^g \alpha(f, X_2).id$.

2. Suppose $\langle A, v\rangle \in \alpha(f, X_1 \times^g X_2).AVP$. Then there exists a $\langle A, v'\rangle \in X_1 \times^g X_2$ such that $v = f(v')$. One of three conditions must now hold:

   - $\langle A, v'\rangle \in X_1.AVP$ and no attribute value pair of the form $\langle A, -\rangle$ exists in $X_2.AVP$. In this case, $\langle A, f(v')\rangle = \langle A, v\rangle \in \alpha(f, X_1)$ and there is no pair of the form $\langle A, -\rangle \in \alpha(f, X_2)$. It follows that
     $\langle A, v\rangle \in \alpha(f, X_1) \times^g \alpha(f, X_2).AVP$.
   - $\langle A, v'\rangle \in X_2.AVP$ and no attribute value pair of the form $\langle A, -\rangle$ exists in $X_1.AVP$. This case is the mirror image of the previous case.
   - In this case, we have $\langle A, v_1\rangle \in X_1.AVP$ and $\langle A, v_2\rangle \in X_2.AVP$ and $g(A, \{v_1, v_2\} = v'$. By definition, we know that $\langle A, f(v_1) \in \alpha(f, X_1).AVP$ and $\langle A, f(v_2) \in \alpha(f, X_2).AVP$. By definition of CPRODUCT , it follows that $\langle A, g(A, \{f(v_1, f(v_2)\})\rangle \in \alpha(f, X_1) \times^g \alpha(f, X_2).AVP$. As f, g are compatible, we know that , $g(A, \{f(v_1, f(v_2)\}) = g(A, \{v_1, v_2\})$ which means that
     $\langle A, v\rangle \in \alpha(f, X_1) \times^g \alpha(f, X_2)$.

   The reverse inclusion may be proved in a similar manner.

Thus, when f is sensible and when certain conditions hold, APPLY may be pushed inside a CPRODUCT . In ordinary relational databases, we can push selection inside a Cartesian Product. This is also possible in PowerPoint databases as long as the selection condition does not involve common attributes.

Theorem 2.3.8 (Pushing SELECT Inside CPRODUCT) *Suppose g is an attribute merge function, X_1 and X_2 are both objects (or both slides, or both presentations), and C is a conjunctive selection condition. Furthermore, suppose $C = C_1 \wedge C_2 \wedge C_3$*

where C_1 only mentions attributes in X_1 and C_2 only mentions attributes in X_2 and C_3 mentions attributes in both. Then $\sigma_C(X_1 \times^g X_2) = \sigma_{C_3}(\sigma_{C_1}(X_1) \times^g \sigma_{C_2}(X_2))$.

Proof of Theorem 2.3.8. We show that each of the components of $\sigma_C(X_1 \times^g X_2)$ and $\sigma_{C_3}(\sigma_{C_1}(X_1) \times^g \sigma_{C_2}(X_2))$ are identical.

1. $\sigma_C(X_1 \times^g X_2).id = X_1.id \cup X_2.id = \sigma_{C_3}(\sigma_{C_1}(X_1) \times^g \sigma_{C_2}(X_2)).id$.

2. Suppose $\langle A, v\rangle \in \sigma_C(X_1 \times^g X_2)$. This means that $\langle A, v\rangle$ satisfies condition C. As $\langle A, v\rangle \in \sigma_C(X_1 \times^g X_2)$, it means that one of three cases arises:

   - $\langle A, v\rangle \in X_1.AVP$ and no attribute value pair of the form $\langle A, -\rangle$ exists in $X_2.AVP$. In this case, it is clear that $\langle A, v\rangle$ satisfies C_1. Furthermore, there is no attribute value pair of the form $\langle A, -\rangle$ in $X_2.AVP$. Hence, $\langle A, v\rangle \in \sigma_{C_3}(\sigma_{C_1}(X_1) \times^g \sigma_{C_2}(X_2))$.
   - $\langle A, v\rangle \in X_2.AVP$ and no attribute value pair of the form $\langle A, -\rangle$ exists in $X_1.AVP$. This case is the mirror image of the preceding case.
   - In this case, we have $\langle A, v_1\rangle \in X_1.AVP$ and $\langle A, v_2\rangle \in X_2.AVP$ and $g(A, \{v_1, v_2\} = v$. Clearly, $\langle A, v_1\rangle$ satisfies C_1, C_3 and $\langle A, v_2\rangle$ satisfies C_2, C_3. It follows immediately that $\langle A, v\rangle \in \sigma_{C_3}(\sigma_{C_1}(X_1) \times^g \sigma_{C_2}(X_2))$.

   The reverse inclusion may be proved in a similar manner.

2.3.6 The JOIN Operator

I define the important concept of a JOIN , generalizing the "conditional join" in relational databases. Let us first define an object join condition.

Definition 2.3.21 (Object Join Condition (OJC)) *An object condition C is called an* object join condition *iff it has two root variables, denoted $O_1, O_2 \in V_o$ respectively, in it.*

Without loss of generality, we will always assume that the variables in an OJC are called O_1, O_2.

Definition 2.3.22 (JOIN Operator) *Suppose C is an OJC and f is an attribute merge condition.*

1. *The join of two objects o_1, o_2 under f, C, denoted $o_1 \bowtie_C^f o_2$ is defined as $merge_f(o_1, o_2)$ if $C[O_1/o_1, O_2/o_2]$ is true, and undefined otherwise. Here, $merge_f(o_1, o_2)$ merges the objects o_1, o_2's common attributes together in accordance with the attribute merge function f (other attributes are left unchanged). As usual, $C[O_1/o_1, O_2/o_2]$ denotes the condition obtained by replacing all occurrences of O_1 (resp. O_2) in C by o_1 (resp. o_2).*
2. *The join of two slides s_1, s_2 under f, C, denoted $s_1 \bowtie_C^f s_2$ is the slide s such that:*
   - $s.id = s_1.id \cup s_2.id$.
   - $s.Titles = s_1.Titles \cup s_2.Titles$.
   - $s.Objects = \{o \mid o \in (s_1 \times^f s_2).Objects \wedge o_1 \bowtie_C^f o_2$ *is defined* $\}$.
3. *The join of two presentations p_1, p_2 under C, denoted $p_1 \bowtie_C p_2$ is defined similarly:*
   - $p.Titles = p_1.Titles \cup p_2.Titles$.
   - $p.Authors = p_1.Authors \cup p_2.Authors$.
   - $p.Slides = \{s \mid s \in (p_1 \times^f p_2).Slides \wedge s.Objects \neq \emptyset\}$.

Notice that in this case, we must still define succ. *If* $s \in p_1 \bowtie_C p_2$*, then we say that* $\mathsf{succ}(s) = s'$ *where* $s' \in p_1 \bowtie_C p_2.Slides$ *and there is an integer* i *such that* $s' = \mathsf{succ}^i_{prod}(s)$ *where* succ_{prod} *denotes the successor relationship in* $p_1 \times^f p_2$ *and for all* $1 \leq j < i$*,* $\mathsf{succ}^j_{prod}(s)$ *is not in* $p_1 \bowtie_C p_2.Slides$*.*

Theorem 2.3.9 (Pullback Theorem: APPLY **and** JOIN **)** *Suppose* f *is a sensible object transformation function and* g *is an attribute merge function such that* f, g *are compatible. Suppose* X_1 *and* X_2 *are both objects (or both slides, or both presentations). Then* $\alpha(f, X_1 \bowtie^g X_2) = \alpha(f, X_1) \bowtie^g \alpha(f, X_2)$*.*

Proof of Theorem 2.3.9. Similar to the proof of Theorem 2.3.7.

The following theorem on pushing selection inside join is similar to that for pushing selection inside a Cartesian Product.

Theorem 2.3.10 (Pushing SELECT **Inside** JOIN **)** *Suppose* f *is an object transformation function and* g *is an attribute merge function. Suppose* X_1 *and* X_2 *are both objects (or both slides, or both presentations) and suppose* C *is a conjunctive selection condition. Furthermore, suppose* $C = C_1 \wedge C_2 \wedge C_3$ *where* C_1 *only mentions attributes in* X_1 *and* C_2 *only mentions attributes in* X_2 *and* C_3 *mentions attributes in both, and* D *is an OJC. Then* $\sigma_C(X_1 \bowtie^g_D X_2) = \sigma_{C_3}(\sigma_{C_1}(X_1) \bowtie^g_D \sigma_{C_2}(X_2))$*.*

Proof of Theorem 2.3.10. Similar to the proof of Theorem 2.3.8.

2.3.7 Set Operators

The *union* of two presentations is (intuitively) the result of combining their slides together. The *intersection* of two presentations restricts the result to slides that are present in both presentations. Finally, the *difference* of two presentations returns slides that are present in the first presentation but do not occur in the second one.

However, all these operations assume that we have a mechanism for determining when a slide is "equal" to another slide. In practice, however, we may consider two slides to be equivalent in many different cases. For example, two slides may visually look the same but may have different sets of authors - this could arise in the case of plagiarism for instance. Alternatively, they may have the same content, but may use different colors or fonts. Should the slides involved be considered the same under these two conditions?

I allow the user to make this decision. Accordingly, set operations all use a binary relation $\sim$ on slides. This binary relation determines when two slides are considered equivalent. The user can execute a set operation *under* a given binary relation $\sim$. In the system, this would correspond to selecting an option from a given menu of choices and executing queries under that option. The $\sim$ operator is not required to be an equivalence relation at this point - it will be required later for some of the equivalence theorems to work.

Definition 2.3.23 (Presentation Set Operators) *The union of two presentations p_1, p_2 with respect to " $\sim$ " is a new presentation $p_1 \cup_\sim p_2$ such that*

$$\begin{aligned}
(p_1 \cup_\sim p_2).id &= p_1.id \cup p_2.id \\
(p_1 \cup_\sim p_2).Titles &= p_1.Titles \cup p_2.Titles \\
(p_1 \cup_\sim p_2).Authors &= p_1.Authors \cup p_2.Authors \\
(p_1 \cup_\sim p_2).Slides &= p_1.Slides \cup \{s \mid s \in p_2.Slides \wedge (\forall s' \in p_1.Slides)\ s \not\sim s'\} \\
(p_1 \cup_\sim p_2).succ &= \mathsf{succ}'
\end{aligned}$$

The intersection of p_1, p_2 with respect to " $\sim$ " is a new presentation $p_1 \cap_\sim p_2$ such that

$$\begin{aligned}
(p_1 \cap_\sim p_2).Titles &= p_1.Titles \cap p_2.Titles \\
(p_1 \cap_\sim p_2).Authors &= p_1.Authors \cap p_2.Authors
\end{aligned}$$

$$
\begin{aligned}
(p_1 \cap_\sim p_2).id &= p_1.id \cap p_2.id \\
(p_1 \cap_\sim p_2).Slides &= \{s \mid s \in p_1.Slides \wedge (\exists s' \in p_2.Slides)\ s \sim s'\} \\
(p_1 \cap_\sim p_2).succ &= \mathsf{succ}'
\end{aligned}
$$

Finally, the difference of p_1, p_2 with respect to " $\sim$ " is a new presentation $p_1 -_\sim p_2$ such that

$$
\begin{aligned}
(p_1 -_\sim p_2).id &= p_1.id - p_2.id \\
(p_1 -_\sim p_2).Titles &= p_1.Titles - p_2.Titles \\
(p_1 -_\sim p_2).Authors &= p_1.Authors - p_2.Authors \\
(p_1 -_\sim p_2).Slides &= \{s \mid s \in p_1.Slides \wedge (\forall s' \in p_2.Slides)\ s \not\sim s'\} \\
(p_1 -_\sim p_2).succ &= \mathsf{succ}'
\end{aligned}
$$

In all three cases, succ' is defined so that the result retains ordering of the slides in both presentations and slides from p_1 occur before slides from p_2.

Set operators on slides can be defined in a similar manner.

Definition 2.3.24 (Slide Set Operators) *Suppose there is a binary relation " $\sim$ " defined on objects. The union of two slides s_1, s_2 with respect to " $\sim$ " is a new slide $s_1 \cup_\sim s_2$ such that*

$$
\begin{aligned}
(s_1 \cup_\sim s_2).id &= s_1.id \cup s_2.id \\
(s_1 \cup_\sim s_2).Titles &= s_1.Titles \cup s_2.Titles \\
(s_1 \cup_\sim s_2).Objects &= s_1.Objects \cup \{o \mid o \in s_2.Objects \wedge (\forall o' \in s_1.Objects)\ o \not\sim o'\}
\end{aligned}
$$

The intersection of s_1, s_2 with respect to " $\sim$ " is a slide $s_1 \cap_\sim s_2$ such that

$$
\begin{aligned}
(s_1 \cap_\sim s_2).id &= s_1.id \cap s_2.id \\
(s_1 \cap_\sim s_2).Titles &= s_1.Titles \cap s_2.Titles \\
(s_1 \cap_\sim s_2).Objects &= \{o \mid o \in s_1.Objects \wedge (\exists o' \in s_2.Objects)\ o \sim o'\}
\end{aligned}
$$

Finally, the difference of s_1, s_2 with respect to " $\sim$ " is a new slide $s_1 -_\sim s_2$ such that

$$\begin{aligned}(s_1 -_\sim s_2).id &= s_1.id - s_2.id \\ (s_1 -_\sim s_2).Titles &= s_1.Titles - s_2.Titles \\ (s_1 -_\sim s_2).Objects &= \{o \mid o \in s_1.Objects \wedge (\forall o' \in s_2.Objects)\, o \not\sim o'\}\end{aligned}$$

Unlike the classical relational algebra case, pushing selection inside set operators requires that the binary relation $\sim$ satisfy some additional conditions. Given an object (resp. slide, presentation) selection condition C, we say that C is $\sim$-*compatible* iff for all objects o_1, o_2 (resp. slides s_1, s_2, presentations p_1, p_2) if $o_1 \sim o_2$ (resp. $s_1 \sim s_2$, $p_1 \sim p_2$) then $C[O/o_1]$ holds iff $C[O/o_2]$ holds (resp. $C[S/s_1]$ holds iff $C[S/s_2]$ holds, $C[P/p_1]$ holds iff $C[P/p_2]$ holds).

The following theorem says that whenever selection conditions are $\sim$-compatible, we may push selection inside set operators.

Theorem 2.3.11 (Pushing SELECT Inside Set Operators) *Suppose X_1, X_2 are both slides (presentations) and x_1, x_2 are both objects (slides). Suppose C is a selection condition that is $\sim$-compatible. Then:*

$$\begin{aligned}\sigma_C(X_1) \cup_\sim \sigma_C(X_2) &= \sigma_C(X_1 \cup_\sim X_2), \\ \sigma_C(X_1) \cap_\sim \sigma_C(X_2) &= \sigma_C(X_1 \cap_\sim X_2), \\ \sigma_C(X_1) -_\sim \sigma_C(X_2) &= \sigma_C(X_1 -_\sim X_2).\end{aligned}$$

Proof of Theorem 2.3.11. We show that $\sigma_C(X_1) \cup_\sim \sigma_C(X_2) = \sigma_C(X_1 \cup_\sim X_2)$ when X_1, X_2 are slides. The other equivalences have similar proofs.

Suppose $o \in \sigma_C(X_1) \cup_\sim \sigma_C(X_2)$. In this case, o satisfies C and either $o \in \sigma_C(X_1)$ or $o \in \sigma_C(X_2)$ and there is no $o' \in \sigma_C(X_1)$ such that $o' \sim o$. But then $o \in X_1 \cup_\sim X_2$ and as o satisfies C, it follows that $o \in \sigma_C(X_1 \cup_\sim X_2)$. The reverse inclusion is similar.

Intuitively, $\sim$-compatibility is necessary for the above theorem because it is possible for two objects to satisfy $o_1 \sim o_2$ but we may not have both objects satisfy a given selection condition. Note that the above theorem does not even require $\sim$ to be an equivalence relation.

Likewise, we are interested in the possibility of pushing the APPLY operator inside a set operation. Again, this can be done only under some conditions that $\sim$ and the transformation function need to satisfy. We say an object (resp. slide, presentation) transformation function f *preserves* $\sim$ iff $\forall x \forall y \; x \sim y \leftrightarrow f(x) \sim f(y)$. The following theorem says that in this case, we may push the APPLY operation inside a set operation.

Theorem 2.3.12 (Pushing APPLY Inside Set Operators) *Suppose X_1, X_2 are both slides (presentations) and x_1, x_2 are both objects (slides). Suppose f is a transformation function that is $\sim$-compatible. Then:*

$$\begin{aligned}
\alpha(X_1, f) \cup_\sim \alpha(X_2, f) &= \alpha(X_1 \cup_\sim X_2, f), \\
\alpha(X_1, f) \cap_\sim \alpha(X_2, f) &= \alpha(X_1 \cap_\sim X_2, f), \\
\alpha(X_1, f) -_\sim \alpha(X_2, f) &= \alpha(X_1 -_\sim X_2, f).
\end{aligned}$$

Proof of Theorem 2.3.12. We show that $\alpha(X_1, f) \cap_\sim \alpha(X_2, f) = \alpha(X_1 \cap_\sim X_2, f)$ when X_1, X_2 are slides. Proofs of the other equivalences are similar.

Suppose $o \in \alpha(X_1, f) \cap_\sim \alpha(X_2, f)$. Then $o \in \alpha(X_1, f)$ and there exists an $o' \in \alpha(X_2, f)$ such that $o \sim o'$. As $o \in \alpha(X_1, f)$, there is an $o_1 \in X$ such that $o = f(o_1)$. Likewise, there is an $o_2 \in X_2$ such that $o = f(o_2)$. As f is strongly $\sim$-preserving, $o_1 \sim o_2$. Hence, $o_1 \in X_1 \cup_\sim X_2$ and so $o = f(o_1) \in \alpha(X_1 \cup_\sim X_2, f)$. The reverse inclusion is similar.

Again, without the restriction of $\sim$-preservation, the above theorem would be false.

Last, but not least, let us consider whether projection can be pushed inside set operators. A set $\mathcal{A}'$ of attributes is said to be *$\sim$-preserving* iff for all objects o_1, o_2 it is the case that $o_1 \sim o_2 \leftrightarrow o_1' \sim o_2'$ where o_1', o_2' denote the restriction of o_1, o_2 respectively to the attributes in $\mathcal{A}'$. The following theorem says that when objects (resp. slides, presentations) are $\sim$-preserving w.r.t. $\mathcal{A}$, then we can push projection into set operations.

Theorem 2.3.13 (Pushing PROJECT Inside Set Operators) *Suppose X_1, X_2 are both slides (presentations) and x_1, x_2 are both objects (slides). Further suppose that $\mathcal{A}'$ is $\sim$-preserving. Then:*

$$\begin{aligned}
\pi_{\mathcal{A}'}(X_1) \cup_\sim \pi_{\mathcal{A}'}(X_2) &= \pi_{\mathcal{A}'}(X_1 \cup_\sim X_2), \\
\pi_{\mathcal{A}'}(X_1) \cap_\sim \pi_{\mathcal{A}'}(X_2) &= \pi_{\mathcal{A}'}(X_1 \cap_\sim X_2), \\
\pi_{\mathcal{A}'}(X_1) -_\sim \pi_{\mathcal{A}'}(X_2) &= \pi_{\mathcal{A}'}(X_1 -_\sim X_2).
\end{aligned}$$

Proof of Theorem 2.3.13. We show the proof of the equivalence $\pi_{\mathcal{A}'}(X_1) \cap_\sim \pi_{\mathcal{A}'}(X_2) = \pi_{\mathcal{A}'}(X_1 \cap_\sim X_2)$ in the case when X_1, X_2 are objects.

Suppose $o \in \pi_{\mathcal{A}'}(X_1) \cap_\sim \pi_{\mathcal{A}'}(X_2)$. In this case, there is an object $o_1 \in \pi_{\mathcal{A}'}(X_1)$ and an object $o_2 \in \pi_{\mathcal{A}'}(X_2)$ such that:

- o_1's restriction, o_1', to the attributes in $\mathcal{A}'$ is o and
- $o_1' \sim o_2'$ where o_2' is the restriction of some object in X_2 to the attributes in $\mathcal{A}'$.

As $\mathcal{A}'$ is $\sim$-preserving, it follows that $o_1 \sim o_2$. But then $o_1 \in X_1 \cap_\sim X_2$ and the retriction to o_1 to the attributes in $\mathcal{A}'$ is o - thus, $o \in \pi_{\mathcal{A}'}(X_1 \cap_\sim X_2)$. The reverse inclusion is similar.

The above theorem says that as long as $\mathcal{A}'$ is $\sim$-preserving, we can push PROJECT inside the set operations.

2.4 Implementation and Cost Model

I have implemented the pptA algebra and query optimizer in C++ on top of Oracle. My implementation, shown in Figure 2.3, also includes a command line interface for issuing queries and a web-based GUI (shown in Figure 2.4) written in PERL that helps the user state his query, executes the query, and renders its result to a sequence of JPEG images on a webpage. The system consists of over 12,500 lines of code.

For converting batches of PPT files into a relatively format-neutral representation stored in the database, I wrote a Visual Basic for Applications (VBA) program that accesses PowerPoint via Microsoft's COM API. Using the raw COM API or the "native COM support" from Visual C++ proved to be a daunting (and eventually impossible) task for us. This VBA program makes it possible to directly read all PPT files in a given directory and populate the database with essential information about what objects, slides, and presentations.

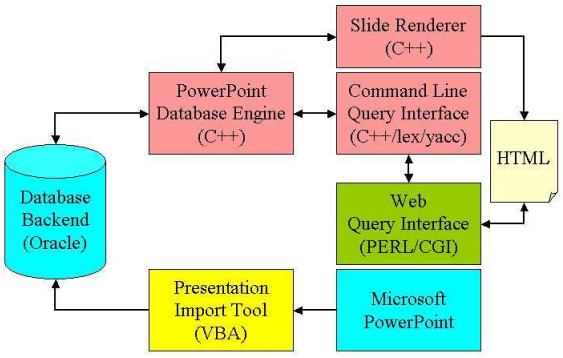


Figure 2.3: System Architecture.

The algebraic engine is implemented as a framework of C++ classes. The most important classes are `Object`, `Slide`, and `Show` that represent objects, slides, and presentations respectively. An object[3] in these classes can represent either a real entity present in the database (so called *explicit* object) or a query (i.e. an *implicit* object). A query may also be explicitly materialized and stored in the database. In this case, the implicit object is said to be *materialized*. Any implicit object can be materialized with the `Create()` function call. Implicit object materializations are deleted from the database when the corresponding objects are destroyed. To keep the query materialization, the user may want to call the `Clone()` function on an object. For implicit objects, this function will execute the query if needed, detach the materialization and keep it as an independent entity in the database. For explicit objects, the `Clone()` call will simply make a copy of the entity.

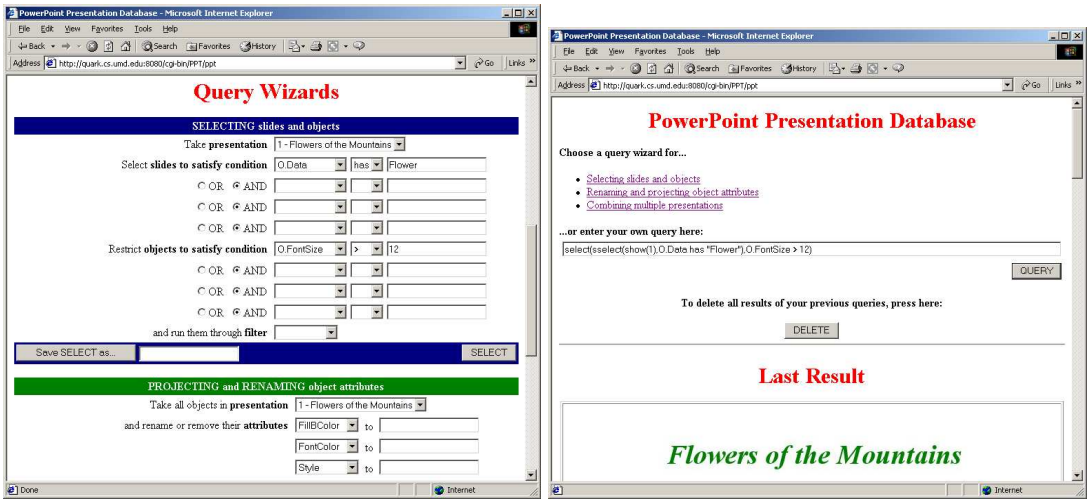


Figure 2.4: Web-Based GUI.

All algebraic operators are implemented as C++ routines containing embedded SQL statements. These programs are quite complex and may involve higher order

[3]In this paragraph, we are talking about C++ objects, not the pptA objects.

functions (e.g. APPLY is a higher order function), external function calls (e.g. in the case of cartesian product and JOIN where the operator is parameterized by a merge strategy, as well as in the case of the set operations where a binary $\sim$ relation is invoked). To build a cost model for such queries, we needed to cost these operations. My cost estimates build upon traditional relational cost models, but account for the C++ implementations of the pptA operators which are not accounted for in relational algebra.

Let us associate two estimated quantities with each algebraic operation. I() is the number of items (objects, slides, or presentations) returned and C() is the cost estimate (in milliseconds of execution time). Table 2.1 shows the cost model for pptA operators on slides. The models for object and presentation operators are very similar. The constants of the form k_{xxx} are dependent on the performance of the Oracle DBMS performance. Thus, k_{del} is the average time to delete an object, k_{s1} is the average time required to evaluate a selection condition on a single object, and so forth. Constants of the form n_{xxx} represent the typical selectivity of the set operators on the given data. Thus, n_u is the average percentage of objects copied from the second input by the union operation, etc.

Values for all these constants were obtained by running multiple pptA queries on the engine compiled with calibration code that I wrote. After observing the behavior of about 11,100 queries on the data generated for the experiments, I computed the values shown in Table 2.2.

To verify reliability of the cost model, I ran a query corresponding to each elementary operation on 100 random slides while measuring execution times. Several runs with different selectivities were made for queries that involve selection. I then compared cumulative execution time for each query type with the estimated time ob-

$$
\begin{aligned}
\mathsf{I}(s) &= \textit{number of objects in slide } s \\
\mathsf{C}(s) &= \textit{cost to create/copy slide } s \textit{ (milliseconds)} \\
\mathsf{C}(f) &= \textit{cost to execute function } f \textit{ (milliseconds)} \\
sel(C) &= \textit{selectivity of } C \textit{ ([0; 1])} \\
\mathsf{I}(\pi_{\overline{A}}(s)) &= \mathsf{I}(s) \\
\mathsf{C}(\pi_{\overline{A}}(s)) &= \mathsf{C}(s) + k_{p0} + k_{p3} \cdot \mathsf{I}(s) \cdot card(A) \\
\mathsf{I}(\rho^r(s)) &= \mathsf{I}(s) \\
\mathsf{C}(\rho^r(s)) &= \mathsf{C}(s) + k_{r0} + k_{r3} \cdot \mathsf{I}(s) \cdot \mathsf{C}(r) \\
\mathsf{I}(\sigma_C(s)) &= \mathsf{I}(s) \cdot sel(C) \\
\mathsf{C}(\sigma_C(s)) &= \mathsf{C}(s) + k_{s0} + k_{s1} \cdot \mathsf{I}(s) + k_{del} \cdot \mathsf{I}(s) \cdot (1 - sel(C)) \\
\mathsf{I}(\alpha(s, f)) &= \mathsf{I}(s) \\
\mathsf{C}(\alpha(s, f)) &= \mathsf{C}(s) + k_{a0} + k_{a1} \cdot \mathsf{I}(s) + \mathsf{I}(s) \cdot \mathsf{C}(f) \\
\mathsf{I}(s_1 \bowtie_C^f s_2) &= \mathsf{I}(s_1) \cdot \mathsf{I}(s_2) \cdot sel(C) \\
\mathsf{C}(s_1 \bowtie_C^f s_2) &= \mathsf{C}(s_1) + \mathsf{C}(s_2) + k_{j0} + k_{j1} \cdot \mathsf{I}(s_1) + k_{j2} \cdot \mathsf{I}(s_2) \\
&+ \quad + k_{j3} \cdot \mathsf{I}(s_1) \cdot \mathsf{I}(s_2) + \mathsf{I}(s_1) \cdot \mathsf{I}(s_2) \cdot sel(C) \cdot \mathsf{C}(f) \\
\mathsf{I}(s_1 \cup_\sim s_2) &= n_u \cdot \mathsf{I}(s_1) \cdot \mathsf{I}(s_2) \\
\mathsf{C}(s_1 \cup_\sim s_2) &= \mathsf{C}(s_1) + \mathsf{C}(s_2) + k_{u0} + k_{u1} \cdot \mathsf{I}(s_1) + k_{u2} \cdot \mathsf{I}(s_2) + k_{u3} \cdot n_u \cdot \mathsf{I}(s_2) \\
\mathsf{I}(s_1 \cap_\sim s_2) &= min(\mathsf{I}(s_1) \cdot (1 - n_i), \mathsf{I}(s_2)) \\
\mathsf{C}(s_1 \cap_\sim s_2) &= \mathsf{C}(s_1) + \mathsf{C}(s_2) + k_{i0} + k_{i1} \cdot \mathsf{I}(s_1) + k_{i2} \cdot \mathsf{I}(s_2) \\
&+ \quad + k_{i3} \cdot \mathsf{I}(s_2) \cdot min(\mathsf{I}(s_1) \cdot (1 - n_i), \mathsf{I}(s_2)) k_{del} \cdot max(\mathsf{I}(s_1) \cdot n_i, \mathsf{I}(s_1) - \mathsf{I}(s_2)) \\
\mathsf{I}(s_1 -_\sim s_2) &= \mathsf{I}(s_1) - min(\mathsf{I}(s_1) \cdot n_d, \mathsf{I}(s_2)) \\
\mathsf{C}(s_1 -_\sim s_2) &= \mathsf{C}(s_1) + \mathsf{C}(s_2) + k_{d0} + k_{d1} \cdot \mathsf{I}(s_1) + k_{d2} \cdot \mathsf{I}(s_2) \\
&+ \quad + k_{d3} \cdot \mathsf{I}(s_2) \cdot (\mathsf{I}(s_1) - 0.5 \cdot min(\mathsf{I}(s_1) \cdot n_d, \mathsf{I}(s_2))) k_{del} \cdot max(\mathsf{I}(s_1) \cdot n_d, \mathsf{I}(s_1) - \mathsf{I}(s_2))
\end{aligned}
$$

Table 2.1: Cost Model for Slide Operations.

tained from the cost model. The results, shown in Figure 2.5, confirm that the cost model produces estimates sufficiently close to the real execution times

k_{del}	27.6135	n_u	0.437376	n_i	0.212705	n_d	0.787295
k_{a0}	12.2499	k_{a1}	9.19035	k_{s0}	12.5987	k_{s1}	9.69176
k_{p0}	12.5024	k_{p3}	3.57006	k_{r0}	12.5518	k_{r3}	3.66911
k_{j0}	33.02	k_{j1}	2.0733	k_{j2}	0.00133	k_{j3}	22.4299
k_{u0}	30.8862	k_{u1}	12.4178	k_{u2}	0.00486	k_{u3}	90.524
k_{i0}	30.0508	k_{i1}	12.3043	k_{i2}	0.005	k_{u3}	5.60735
k_{d0}	31.1229	k_{d1}	12.3158	k_{d2}	0.00484	k_{d3}	4.92384

Table 2.2: Cost Coefficients.

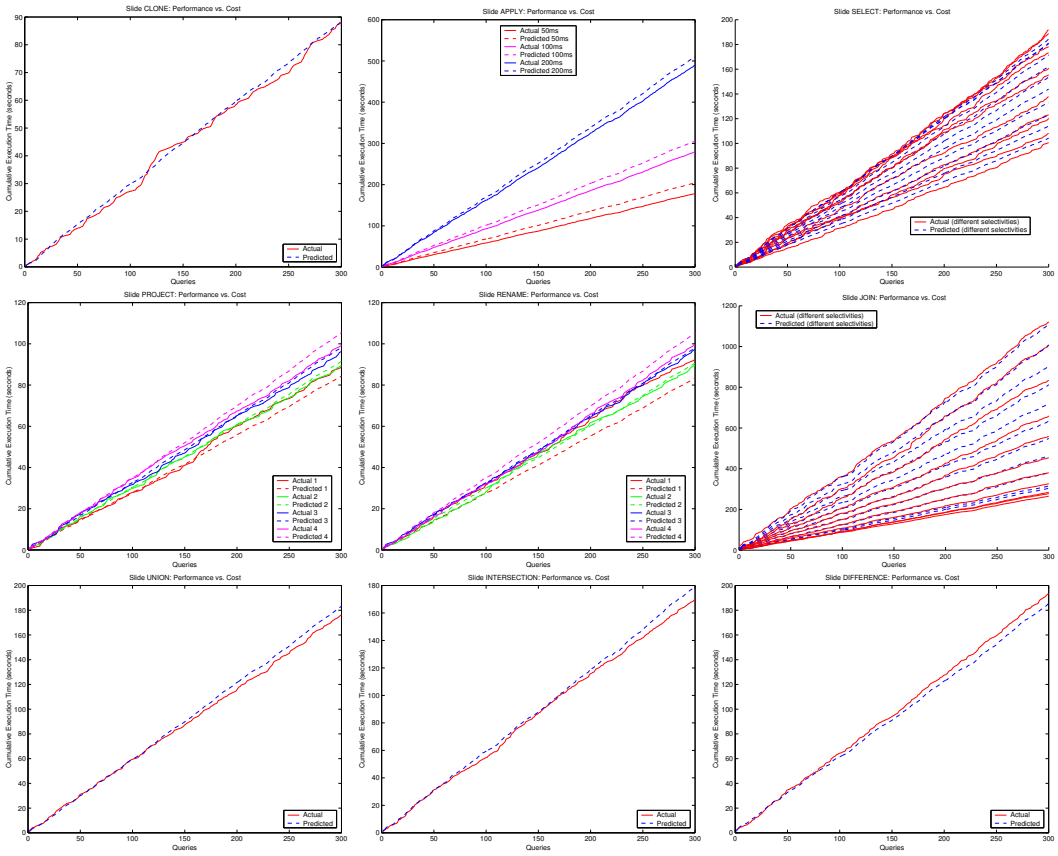


Figure 2.5: Cost Estimation Experiments.

2.5 Experimental Results

I conducted three series of experiments to (i) validate the cost model (as described in the previous section), (ii) identify useful query equivalences, and (iii) assess the value

of the query optimization. This section describes the last two series of experiments.

To determine how the equivalence results of the preceding sections may be used to speed up query processing, I measured the performance of queries on simulated data. I ran the same query on 40 different slides, computing mean execution times, and plotting them as a function of query selectivity (10-100% of all objects on a slide). Each slide followed one of 8 widely used templates (shown in Figure 2.6) and contained 6-11 objects with about 7 attributes each.

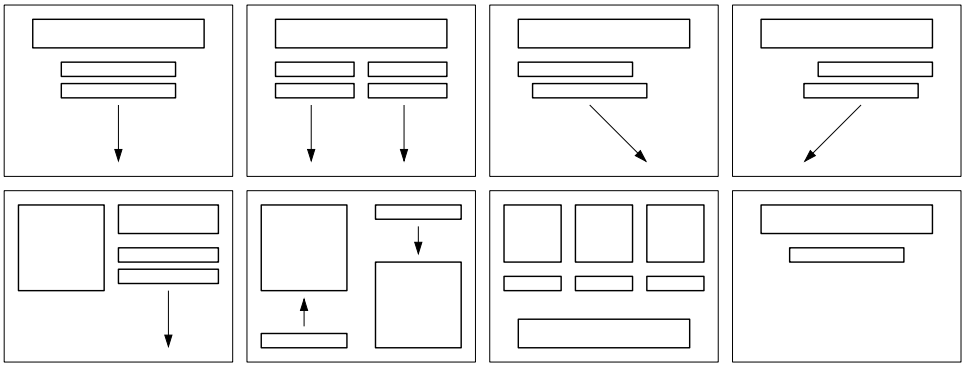


Figure 2.6: Simulated Slide Layouts.

Order of Select and Apply. Figure 2.7 shows execution times for $\sigma_C(\alpha(s, f))$ and $\alpha(\sigma_C(s), f)$ queries, where s is a slide, f is an object transformation function, and C is an object selection condition that selects a given percentage of objects on a slide. Three pairs of graphs are shown — these correspond to the cases when f takes $50ms$, $100ms$, and $200ms$ to execute on each object. As can be seen, pushing SELECT inside APPLY is beneficial especially for lower C selectivities. This can be explained by the lower number of objects that get passed to APPLY . The effect becomes more pronounced as f gets slower: compare the $50ms$ graphs with the $200ms$ graphs.

Order of Apply and Join. Figure 2.8 shows execution times for $\alpha(s_1, f) \bowtie^g_C \alpha(s_2, f)$ and $\alpha(s_1 \bowtie^g_C s_2, f)$ where s_1, s_2 are slides, f is an object trans-

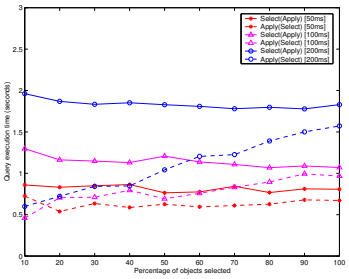


Figure 2.7: Select(Apply) vs. Apply(Select).

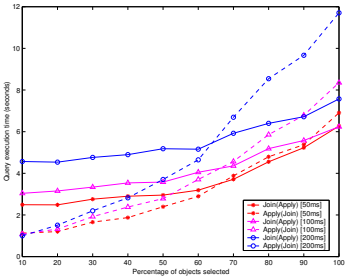


Figure 2.8: Join(Apply) vs. Apply(Join).

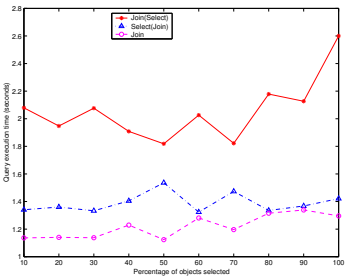


Figure 2.9: Join(Select) vs. Select(Join).

formation function, C is an object selection condition that selects a given percentage of objects from both input slides, and g is an object merge function that just returns the first of its arguments. Three pairs of graphs are shown corresponding to f taking $50ms$, $100ms$, or $200ms$ to execute on each object. The results confirm that pushing APPLY inside JOIN lowers query complexity from quadratic to linear, as APPLY gets to run on $2n$ objects as opposed to n^2. Nevertheless, for lower C selectivities, having APPLY outside is better as JOIN only produces a fraction of n^2 objects. Thus, the choice of a query rewrite must depend on both the estimated selectivity of C and the expected execution time of f.

Order of Select and Join. Finally, Figure 2.9 shows execution times for $\sigma_{C_2}(s_1) \bowtie^g_{C_1} \sigma_{C_2}(s_2)$, $\sigma_{C_2}(s_1 \bowtie^g_{C_1} s_2)$, and $s_1 \bowtie^g_{C_1 \wedge C_2[O/O_1] \wedge C_2[O/O_2]} s_2$ where s_1, s_2 are slides, C_1 is an object join condition, C_2 is an object selection condition, and g is an object merge function that just returns the first of its arguments. The C_1 and C_2 are chosen so that they select a given percentage of objects from both input slides. The results confirm that pushing SELECT inside JOIN speeds things up. Even better results are achieved by pushing SELECT condition into the JOIN condition though - in other words, if we can take the selection condition and infer a new join condition,

then we often get even better results.

Benefits of Query Optimization. The next batch of experiments was used to observe the difference between executing an original query and its optimized equivalent. A set of 77 different queries were run on 100 random slides and real execution times of both the original and the optimized queries were measured. The resulting execution times are shown in Figure 2.11. The times shown in this graph include both the time required for query optimization and the time taken to execute the optimized query. The benefits of query optimization are clearly visible from the graph.

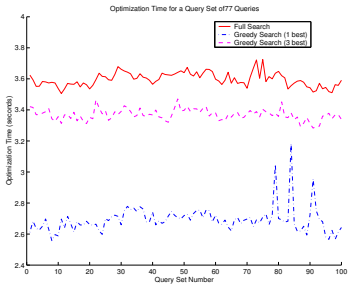


Figure 2.10: Optimization Times.

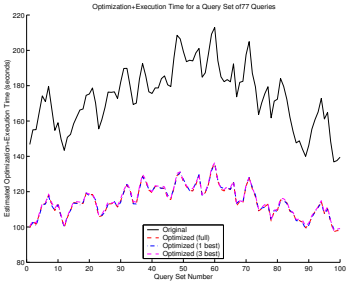


Figure 2.11: Estimated Query Execution Times.

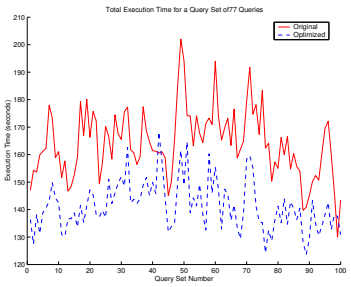


Figure 2.12: Real Query Execution Times.

Figure 2.11 shows estimated execution times for both original query sets and their optimized equivalents. From this figure, it appears that all optimization algorithms produce close results. Figure 2.10 shows that although their running time is different, all three optimization algorithms seem to consume negligently small time in comparison with the actual query execution. Finally, Figure 2.12 compares the *real* execution times of the non-optimized and optimized queries (including the optimization time). The benefits of optimization can be clearly seen at this figure.

2.6 Related Work

There have been two major efforts to date on querying multimedia presentations due to Lee et. al. [49, 46] and Adali et. al.[3]. Both these efforts attempt to model the *playout* of a presentation rather than the physical *layout* of the presentation. In [46], nodes in a graph are media streams, and edges denote a precedence ordering. PowerPoint stores data physically using the concepts of presentation, slides, and objects (within a slide) which is exactly what my model captures. This PowerPoint specific representation is quite different from that of [46] (who do not claim to query PowerPoint presentations). Lee et. al. [46] provide an SQL style language (called GVISUAL) and a calculus and prove that the two are equivalent in expressive power. I have not done this in this work. However, this work is the first to propose a cost model, and to evaluate equivalence results on such a cost model.

In [3], each node in a presentation graph is a non-interactive multimedia presentation, and an edge denotes an interaction that transitions from one node to another. In contrast, my model accurately captures the physical structure of PPT presentations whereas [3] manipulates a *logical representation* of a PPT presentation (which is a very different thing when building an implementation). As a consequence, I am able to have a cost model for pptA operations and an implementation that can be used to evaluate rewrite rules — [3] has neither. In addition, [3] allows playouts to be potentially infinite — of course, PowerPoint presentations are always finite and so by focusing on the physical layout of the presentations (rather than the possibly playouts of the presentation as [3] do), I avoid this problem.

2.7 Conclusions

In this chapter, I have presented a formal model of PowerPoint presentations as well as a relational algebra style algebra to query such PPT databases. This algebra includes new and interesting operators (such as the APPLY operator) as well as interesting join operators (that allow multiple presentations being merged to have common attributes whose values may be merged using a join function). I have proven a set of equivalence results within this algebra. I have developed a cost model for PPT presentations and built a prototype implementation of the system that stores and queries PPT presentation databases. I have conducted a detailed set of experiments to evaluate the efficacy of various equivalence results from the point of view of query rewriting.

Chapter 3

An Algebra for Audio

3.1 Introduction

There is an incredibly rich body of audio data available in the world today. This data comes from radio broadcasts, musical performances, educational lectures, sonars used by submarines and ships, and many other sources. Libraries, museums, recording studios, as well as surveillance agencies, have collected large amounts of audio recordings.

The wealth of available audio data naturally brings up the problem of its systematic storage and processing. The processing must, of course, include searching functions, but is not limited to them, as audio data often has to be amplified, attenuated, mixed, cleaned of noise and distortions, and so forth. The storage subsystem has to accommodate these operations by indexing the data in the best possible way, and for that we need to know how the data will be accessed. All these requirements can be addressed by a *common theory* about audio data processing.

However, to date, there is relatively little work on audio databases, and almost no theory for them (with the exception of [45]). Commercial support for audio exists

in Informix (with the MuscleFish datablade) [7, 8] and DB2 (with the audio data extenders) [76]. In research projects, audio is often used as a means for indexing the accompanying video [85, 39]

Existing efforts in audio information processing can be roughly divided into the following topics: (i) data compression (MPEG), (ii) data delivery [29, 31], (iii) speaker and musical composition recognition (Gracenote,[33, 27]), (iv) speech recognition [71], and (v) word, or phrase search [71, 85]. To the best of my knowledge, no attempts were made to develop a framework that would abstract all the different operations people would like to perform on audio data in the same way the relational algebra abstracts operations on tabular data.

A glance at a standard sound processing package, such as CoolEdit, reveals that the most basic operations on audio include (i) changing volume and pitch, (ii) filtering, usually with frequency filters, (iii) mixing sounds, and (iv) adding special effects, such as fade-ins and fade-outs. When turning to music composition software, such as Cakewalk or Cubase, we also find a need to synchronize sounds from different sources (e.g. multiple MIDI channels, or a MIDI score and a waveform recording). Finally, when maintaining a database of audio recordings, there is an obvious need to search these recordings by occurance of certain keywords, phrases, tunes, or volume patterns.

Ideally, an Audio Database Algebra (ADA) must support all the above-mentioned operations in an integrated way that would allow, say, to search for a certain audio fragment and amplify its volume, or mix two instrumental recordings based on the timing information from their MIDI scores. In this chapter, I propose such an algebra and show some equivalences useful for the query optimization. Furthermore, I present several data structures for indexing audio data [26]. The experiments conducted using

these data structures show that they greatly accelerate such basic algebraic operations as selection, mixing, and matching of audio fragments. Finally, I describe a reference ADA system for creating and querying audio databases. The system consists of the algebraic engine (implemented as a library), the command line query interpreter, and the GUI.

Section 3.2 of this chapter defines the basic model for representing audio data. Section 3.3 covers algebraic operators and equivalences. Sections 3.4 and 3.5 deal with algorithms and data structures for indexing audio. Section 3.6 describes the ADA system implementation and the GUI. Section 3.7 discusses other works in audio databases. Finally, Section 3.8 provides some concluding remarks.

3.2 Audio Data Model

Figure 3.1 shows a very simple audio waveform. Though each waveform is played out continuously, it is typically decomposed into a sequence of discrete *quanta*. The example waveform in Figure 3.1 has 38 quanta of 500 microseconds (μs) each. Let us call the playback time of a single quantum ($500\mu s$ in our example) the *period* δt of a waveform. A waveform also has a *length* ℓ (in our example $\ell = 38$) which is the number of quanta.

An audio recording may contain one or more waveforms similar to the one shown in Figure 3.1. For example, a stereo recording may have two synchronized waveforms for the left and right channels. Waveforms may also be accompanied by other data, such as speech transcript, sequence of phonemes, frequency spectrum, MIDI score, or karaoke cues, that are tied to the same time quanta. Thus, audio lends itself to be described as a *collection of synchronized data streams*. To generalize this concept, let

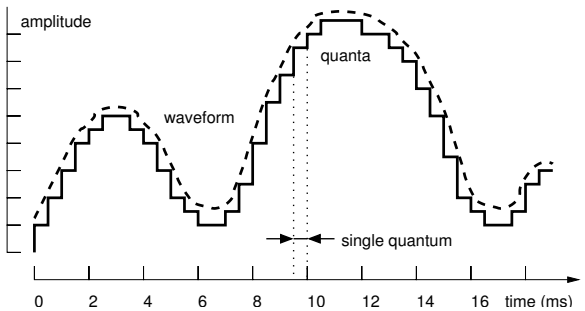


Figure 3.1: Simple Waveform.

us introduce an *audio stream*. Let us assume the existence of some set of types - in addition each type τ has an associated nonempty set $dom(\tau)$ called its *domain*.

Definition 3.2.1 (Audio Stream) *An* audio stream f *of type* τ *and length* ℓ *is an* ℓ*-tuple* $f = \langle v_0, \ldots, v_{\ell-1} \rangle$ *where each* $v_i \in dom(\tau)$.

Each stream f of type τ has a *domain*, $dom(f) = dom(\tau)$, which is a nonempty set. A stream f of length ℓ is a mapping of $[0, \ell)$ to $dom(f)$. In other words, for each quantum q, $f(q) = f.v_q$ describes the value of stream f at the time quantum q.

For example, consider a stream na called *normalized amplitude* that maps $[0, \ell)$ to $[0, 1]$. If $na(5) = 0.8$, then this means that the normalized amplitude of the fifth quantum of stream na is 0.8. Examples of other streams include amplitude (non-normalized), phonemes, and text transcript.

The *amplitude* stream may have $dom(amplitude) = \mathcal{N}$. Similarly, the stream *phonemes* may have a domain that consists of characters corresponding to phonemes. Likewise, the *text transcript* stream may have the set of all strings as its domain.

If $dom(f)$ has greatest lower and least upper bounds (when f is numeric, or has a total ordering imposed on it), we will use $min(f)$ and $max(f)$ to represent these bounds.

Some audio streams may not be available all the time. For example, the text transcript is absent if there is no speech in the audio. For such cases, assume that each stream f's domain has a special value $dv(f) \in dom(f)$ called the *default value* of that stream and used when no actual value is available. In the example with the text transcript, one may assume that $dv(text) = \epsilon$ (empty string).

To represent an audio recording as a whole, let us now define the concept of an *audio file*, that is a collection of several streams with the same length and period:

Definition 3.2.2 (Audio File) *An* audio file a *is a tuple* $a = \langle \ell, \delta t, \{f_1, \ldots, f_n\} \rangle$ *where* $\ell \geq 0$ *is the* length*,* $\delta t > 0$ *is the* period*, and* $f_1, \ldots, f_n$ *are audio streams of length* ℓ*.*

It is important to note that the streams constituting a single audio file may be stored in different *physical files* on disk. Throughout this chapter I talk about *audio files*, unless explicitly stated otherwise.

From the above discussion, it is clear that each audio stream is characterized by a set of parameters such as domain, default value, and so forth. Let us formalize these parameters by defining a *schema*:

Definition 3.2.3 (Audio Schema) *An* audio stream schema *for a stream* f *is a tuple* $\langle dom, dv \rangle$ *where* $dom(f)$ *is the stream* type *or* domain*, and* $dv(f) \in dom(f)$ *is the* default value*. An* audio schema *for a file* a *is a collection of audio stream schemas for all streams in* a*.*

Later on, we will extend the audio stream schema with more parameters, as they become relevant.

Example 3.2.1 (Audio Stream Schema) *Here are some examples of audio stream schemas:*

f	$dom(f)$	$dv(f)$
normalized amplitude	$[0, 1]$	0
frequency spectrum	$2^{\mathcal{N}\times[0,1]}$	$\langle 0, 0\rangle$
bandwidth	$\mathcal{N}$	0
phoneme stream	$\{all\ phonemes, \epsilon\}$	ϵ
text transcript	$\{all\ strings, \epsilon\}$	ϵ

We are finally ready to define an *audio database*:

Definition 3.2.4 (Audio Database) *An* audio database ADB *is a collection of audio files adhering to the same schema.*

3.3 Algebraic Operators

Throughout the rest of this chapter, I assume the existence of an arbitrary but fixed audio schema with respect to which I define all algebraic operators. I will call it the *audio database schema*. But before we look at algebraic operators, let us define the concept of a *selection condition*.

3.3.1 Selection Conditions

Assume the existence of a set C_a of all possible audio files and a set V_a of all variables ranging over C_a. Let us denote *audio constants* (members of C_a) with a small a_* and *audio variables* (members of V_a) with a capital A_*.

Definition 3.3.1 (Term) *(i) Any member of a set τ is a* term *of type τ. (ii) If X is an audio constant or variable, and f is an audio stream of type τ, then $X.f$ is a* term *of type τ.*

For example, $A.na$ is a term (assuming A is an audio variable).

Definition 3.3.2 (Atom) *(i) Given two terms t_1, t_2 and a binary relation $\sim$, $t_1 \sim t_2$ is an* atom. *(ii) $TRUE$ and $FALSE$ are atoms.*

For example, $A.na > 0.7$ is an atom. When applied to a single audio file, this atom is satisfied by all quanta in the file that have a normalized amplitude over 0.7. A formal definition of satisfaction will be presented shortly.

Definition 3.3.3 (Selection Condition) *(i) Any atom is a* selection condition. *(ii) If C is a selection condition and $d \in [0, length(f))$, then* $\mathsf{before}(C, d)$ *and* $\mathsf{after}(C, d)$ *are selection conditions. (iii) If C_1, C_2 are selection conditions then $C_1 \wedge C_2$, $C_1 \vee C_2$, and $\neg C_1$ are also selection conditions.*

For instance, $A.na > 0.7 \wedge A.na < 0.9$ is a selection condition. When applied to a single audio file, this selection condition is satisfied by all quanta in the file that have a normalized amplitude over 0.7 but less than 0.9.

$\mathsf{before}(A.na > 0.7, 5)$ is also a selection condition. When applied to a single audio file, this selection condition is satisfied by all quanta q in the file such that a quantum q' in the file satisfies $A.na > 0.7$ and q' occurs at most 5 quanta before q.

I now define the *valuation* of variable-free terms - this is needed in order to define satisfaction of selection conditions.

Definition 3.3.4 (Valuation) *The* valuation λ *is a function that takes a variable-free τ-term, c, and a time quantum q, and returns a value from τ such that*

$$\begin{aligned} \lambda(a.f, q) &= a.f(q) \\ \lambda(c, q) &= c \end{aligned}$$

Definition 3.3.5 (Satisfaction) *Let us define a function λ that takes a variable-free selection condition C, a time quantum q, and returns either 1 (satisfied) or 0 (not satisfied).*

$$
\begin{aligned}
\lambda(TRUE, q) &= 1 \\
\lambda(FALSE, q) &= 0 \\
\lambda(x \sim y, q) &= \begin{cases} 1 & if\ \lambda(x,q) \sim \lambda(y,q) \\ 0 & otherwise \end{cases} \\
\lambda(\mathsf{before}(C,d), q) &= \begin{cases} 1 & if\ \exists q' \in [q, q+d]\ \lambda(C,q') \\ 0 & otherwise \end{cases} \\
\lambda(\mathsf{after}(C,d), q) &= \begin{cases} 1 & if\ \exists q' \in [q-d, q]\ \lambda(C,q') \\ 0 & otherwise \end{cases} \\
\lambda(C_1 \wedge C_2, q) &= min(\lambda(C_1,q), \lambda(C_2,q)) \\
\lambda(C_1 \vee C_2, q) &= max(\lambda(C_1,q), \lambda(C_2,q)) \\
\lambda(\neg C, q) &= 1 - \lambda(C,q)
\end{aligned}
$$

A variable-free condition C is called satisfied *or* true *at quantum q iff $\lambda(C,q) = 1$. Otherwise, C is* false*.*

A *local* condition is a special kind of a selection condition that can be evaluated at a single instance of time, without knowledge of other instances:

Definition 3.3.6 (Local Selection Condition) *A selection condition C is called* local *if it does not contain any* $\mathsf{before}()$ *or* $\mathsf{after}()$ *statements.*

One important characteristic of a selection condition is its *footprint* — this is a set of all stream names appearing in the condition w.r.t. a variable:

Definition 3.3.7 (Condition Footprint) *Given a condition C and a variable A, the* condition footprint *of C w.r.t. A is a set of stream names $fp_A(C)$ such that:*

$$\begin{aligned}
fp_A(_ \sim A.f) &= \{f\} \\
fp_A(A.f \sim _) &= \{f\} \\
fp_A(A.f_1 \sim A.f_2) &= \{f_1, f_2\} \\
fp_A(\mathsf{before}(C, d)) &= fp_A(C) \\
fp_A(\mathsf{after}(C, d)) &= fp_A(C) \\
fp_A(C_1 \wedge C_2) &= fp_A(C_1) \cup fp_A(C_2) \\
fp_A(C_1 \vee C_2) &= fp_A(C_1) \cup fp_A(C_2) \\
fp_A(\neg C) &= fp_A(C)
\end{aligned}$$

3.3.2 The SELECT Operator

Selection is one of the most basic algebraic operations. For instance, given a recording of a court session, one may want to pick all judge's statements from it. Alternatively, consider the "squelch" function used in miniature voice recorders and shortwave receivers that turns sound on whenever its volume exceeds some threshold. These are just two of many examples where audio selection is useful. When applied to a single audio file, selection chooses all parts of this file that satisfy given condition. When applied to an audio database, it applies the operator to each file and returns the resulting files.

Definition 3.3.8 (SELECT Operator) *Given an audio file a (database* ADB*) and a selection condition C with a single free audio variable A, the* SELECT operator *pro-*

duces a new audio file (database)

$$\begin{aligned}\sigma_C(a) &= \langle a.\ell, a.\delta t, \{f \mid f(q) = \begin{cases} a.f(q) & if\ \lambda(C[A/a], q) = 1 \\ dv(f) & otherwise \end{cases} \}\rangle, \\ \sigma_C(\mathsf{ADB}) &= \{\sigma_C(a) \mid a \in \mathsf{ADB}\}.\end{aligned}$$

Here, $C[A/a]$ denotes the replacement of all occurrences of the audio variable A in C by a.

Thus, the select operator looks at an audio file f and retains all quanta that satisfy the selection condition. Stream values for all other quanta are replaced with the default values. Figure 3.2 shows a simple selection query example $\sigma_{A.na>0.3}(a)$

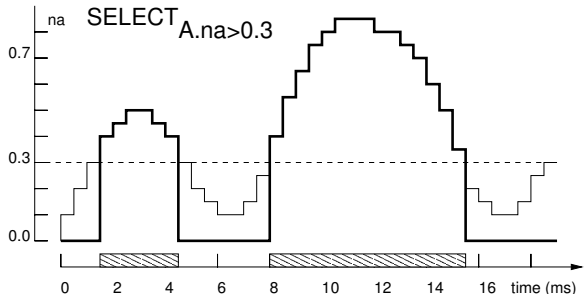


Figure 3.2: SELECT Example.

In a more complex example, consider a court recording a. This file may have a stream $a.speaker$ that identifies the current speaker. Then we can select judge's statements with $\sigma_{A.speaker=judge}(a)$. Alternatively, knowing the base timbre of judge's voice, represented with a set of frequencies F_{judge}, the same operation can be performed with $\sigma_{F_{judge} \subseteq A.freq}(a)$, albeit with lower reliability.

In the instance of a squelch function with threshold k, the selection can be done with a $\sigma_{A.na>k}(a)$ query. To make things more interesting, let us require the squelch to stay on for 500 quanta after the volume falls below the threshold: $\sigma_{\mathsf{after}(A.vol>k,500)}(a)$.

Unlike the relational algebra [80], one cannot arbitrarily change the order of selections in ADA . The order of selection operators can be changed *as long as the selection conditions involved are local selection conditions*.

Theorem 3.3.1 (Swapping SELECT Operators) *Given two* SELECT *operators with local selection conditions C_1, C_2, it is true that*

$$\sigma_{C_1}(\sigma_{C_2}(a)) \;=\; \sigma_{C_2}(\sigma_{C_1}(a)).$$

Proof of Theorem 3.3.1. This proof is similar to the proof of commutativity of the classical relational SELECT . As the theorem requires both C_1 and C_2 to be local selection conditions, one can look at each time quantum independently from all other quanta. Consider then an arbitrary stream f in $a' = \sigma_{C_2}(\sigma_{C_1}(a))$ and $a'' = \sigma_{C_1}(\sigma_{C_2}(a))$ at a time quantum q. By the definition of SELECT we can say that:

1. If $\lambda(C_1[A/a], q) = 0$ and $\lambda(C_2[A/a], q) = 0$ then $a'.f(q) = dv(f)$ and $a''.f(q) = dv(f)$, i.e. $a'.f(q) = a''.f(q)$.

2. If $\lambda(C_1[A/a], q) = 1$ and $\lambda(C_2[A/a], q) = 0$ then $a'.f(q) = \sigma_{C_2}(a).f(q) = dv(f)$ and $a''.f(q) = dv(f)$, i.e. $a'.f(q) = a''.f(q)$.

3. If $\lambda(C_1[A/a], q) = 0$ and $\lambda(C_2[A/a], q) = 1$, the reasoning is similar to (2).

4. Finally, if $\lambda(C_1[A/a], q) = 1$ and $\lambda(C_2[A/a], q) = 1$ then $a'.f(q) = \sigma_{C_2}(a).f(q) = a.f(q)$ and $a''.f(q) = \sigma_{C_1}(a).f(q) = a.f(q)$, i.e. again $a'.f(q) = a''.f(q)$.

We have shown that for any arbitrary stream f and time quantum q, $a'.f(q) = a''.f(q)$. Therefore, $a' = a''$.

3.3.3 The BETWEEN Operator

The BETWEEN operator is similar to the SELECT operator, but has *two* conditions instead of one, called the *start* and *stop* conditions. BETWEEN selects ranges of quanta starting with the start condition being satisfied (or "triggered") and ending with the stop condition being satisfied. For example, one can apply this operator to the text transcript stream to select all speech starting with a given word and ending with a pause.

Definition 3.3.9 (BETWEEN Operator) *Given an audio file a (database* ADB*) and two selection conditions C_1, C_2 each with a single free audio variable A, the* BETWEEN operator *produces a new audio file (database)*

$$\beta^{C_2}_{C_1}(a) = \langle a.\ell, a.\delta t, \{f \mid f(q) = \begin{cases} a.f(q) & if\ \exists q' \in [0,q]\ \lambda(C_1[A/a], q') = 1 \wedge \\ & \forall q'' \in [q',q]\ \lambda(C_2[A/a], q'') = 0 \\ dv(f) & otherwise \end{cases} \}\rangle,$$

$$\beta^{C_2}_{C_1}(\mathsf{ADB}) = \{\beta^{C_2}_{C_1}(a) \mid a \in \mathsf{ADB}\}.$$

Thus, the BETWEEN operator looks at an audio file f and retains ranges of quanta starting with the first condition becoming true and ending with the second condition becoming true. Stream values for all other quanta are replaced with the default values. Figure 3.3 shows a simple example query $\beta^{A.na<0.2}_{A.na>0.3}(a)$

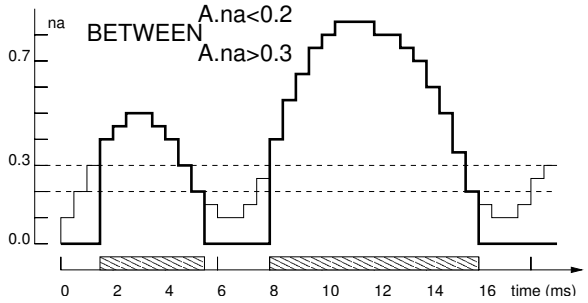


Figure 3.3: BETWEEN Example.

Unfortunately, unlike SELECT operators, BETWEEN operators cannot be swapped in order. Consider two queries: $a' = \beta_{C_3}^{C_4}(\beta_{C_1}^{C_2}(a))$ and $a'' = \beta_{C_1}^{C_2}(\beta_{C_3}^{C_4}(a))$ at a quantum q such that $\beta_{C_1}^{C_2}(a)$ selects a range $[q_1, q_2)$ from a, while $\beta_{C_3}^{C_4}(a)$ selects $[q_3, q_4)$. Consider a case when $q_1 < q_3 < q_4 < q_2$, and all conditions evaluate to false everywhere except the range boundaries. In such a case, a'' contains the $[q_3, q_4)$ range, while a' is empty. Thus, $a' \neq a''$, making the general case of swapping BETWEEN operators impossible.

3.3.4 The APPLY Operator

The APPLY operator allows users to transform audio streams in an arbitrary way. Due to its generality, the APPLY can be used for many purposes, such as volume and stereo balance control, fade-ins and fade-outs.

To simplify things, assume that each APPLY operator only changes a single stream. Several APPLY operators can be used to modify multiple streams. Let us start by defining a *transformation function*:

Definition 3.3.10 (Stream Transformation Function) *Given an audio stream schema f, a* stream transformation function *is a mapping $tr_f : dom(f) \rightarrow dom(f)$ such that $tr_f(dv(f)) = dv(f)$.*

In addition to f, a stream transformation function may use values from other streams and/or the current quantum number. Depending on the values used, one can classify transformation functions as follows:

Definition 3.3.11 (Types of Stream Transformation Functions)

- *A stream transformation function is called* local *iff its inputs are restricted to audio stream values from the current time quantum.*

- *A stream transformation function is called* value-local *iff its inputs are restricted to audio stream values from the current time quantum and the current quantum number.*

- *A stream transformation function is called* time-local *iff its inputs are restricted to sets of audio stream values over some period around the current time quantum.*

As in the case of selection conditions, transformation functions have *footprints*, or sets of streams used to compute the transformed stream. I denote the footprint of a transformation function tr_f by $fp(tr_f)$. Due to the requirement that $tr_f(dv(f)) = dv(f)$, tr_f always uses the original value of f to compute the new value, i.e. it is *always* true that $f \in fp(tr_f)$.

Definition 3.3.12 (APPLY Operator) *Given an audio file a (database ADB) and a stream transformation function tr_f, the* APPLY operator *produces a new audio file (database)*

$$\begin{aligned} \alpha(a, tr_f) &= \langle a.\ell, a.\delta t, \{f' \mid f'(q) = \begin{cases} tr_f(a.f'(q)) & if\ f' = f \\ a.f'(q) & otherwise \end{cases} \}\rangle, \\ \alpha(\mathsf{ADB}, tr_f) &= \{\alpha(a, tr_f) \mid a \in \mathsf{ADB}\}. \end{aligned}$$

The following examples show how to control volume, balance, and create fade-in effects using APPLY .

Example 3.3.1 (Volume and Stereo Balance Control) *One can halve the volume of an audio file a using the following query: $\alpha(a, vol(q) = vol(q)/2)$. Given two streams vol_r and vol_l in a stereo audio file a, one can also change the stereo balance by issuing a query $\alpha(\alpha(a, vol_r(q) = vol_r(q) \cdot bal), vol_l(q) = vol_l(q)/bal)$, where bal con-*

trols the balance. Notice that both volume and stereo balance transformation functions are local.

Example 3.3.2 (Fade-in and Fade-out) *To create a five-second fade-in at the beginning of a recording, one can execute a query* $\alpha(a, vol(q) = vol(q) \cdot min(1, q \cdot a.\delta t/5))$. *The fade-out is created by a similar query:* $\alpha(a, vol(q) = vol(q) \cdot max(0, 1 - q \cdot a.\delta t/5))$. *Notice that both fade-in and fade-out transformation functions are* value-local*, but not* local.

Two APPLY operators can be reordered, under certain assumptions.

Theorem 3.3.2 (Swapping APPLY Operators) *Consider two* APPLY *operators with transformation functions* tr_{f_1}, tr_{f_2}. *If* $f_1 \notin fp(tr_{f_2})$ *and* $f_2 \notin fp(tr_{f_1})$, *then the following holds:*

$$\alpha(\alpha(a, tr_{f_2}), tr_{f_1}) \quad = \quad \alpha(\alpha(a, tr_{f_1}), tr_{f_2}).$$

Proof of Theorem 3.3.2. Consider an arbitrary stream s in $a' = \alpha(\alpha(a, tr_f), tr_g)$ and $a'' = \alpha(\alpha(a, tr_g), tr_f)$ at a time quantum q:

1. If $s \neq f$ and $s \neq g$ then $a'.s(q) = a.s(q)$ and $a''.s(q) = a.s(q)$ by the definition of APPLY . Thus, $a'.s(q) = a''.s(q)$.

2. If $s = f$ then the theorem requires that $s \notin fp(tr_g)$. Because $g \in fp(tr_g)$ by the common property of all transformation functions, we can also say that $s \neq g$. Then $a'.s(q) = tr_f(a.s(q))$ and $a''.s(q) = tr_f(a.s(q))$, i.e. $a'.s(q) = a''.s(q)$.

3. If $s = q$, the reasoning is similar to (2).

We have shown that for any arbitrary stream s and time quantum q, $a'.s(q) = a''.s(q)$. Therefore, $a' = a''$.

APPLY and SELECT can also be reordered, if APPLY does not modify any streams used by SELECT :

Theorem 3.3.3 (Swapping APPLY and SELECT) *Given an* APPLY *operator with a value-local transformation function tr_f and a* SELECT *operator with condition C such that $f \notin fp_A(C)$, the following holds:*

$$\alpha(\sigma_C(a), tr_f) \;=\; \sigma_C(\alpha(a, tr_f)).$$

Proof of Theorem 3.3.3. As the theorem requires that $f \notin fp_A(C)$, we can say that $\lambda(C[A/a], q) = \lambda(C[A/\alpha(a, tr_f)], q)$. Also, because the theorem requires tr_f to be value-local, we can look at each time quantum independently from all other quanta. Let us then consider an arbitrary stream s in $a' = \alpha(\sigma_C(a), tr_f)$ and $a'' = \sigma_C(\alpha(a, tr_f))$ at a time quantum q:

1. If $s \neq f$ then $a'.s(q) = \sigma_C(a).s(q)$ and $a''.s(q) = \sigma_C(a).s(q)$ by the definition of APPLY . Thus, $a'.s(q) = a''.s(q)$.

2. If $s = f$ and $\lambda(C[A/a], q) = 1$ then $a'.s(q) = tr_f(a.s(q))$ and $a''.q(s) = tr_f(a.s(q))$ by the definitions of APPLY and SELECT . Thus, $a'.s(q) = a''.s(q)$.

3. If $s = f$ and $\lambda(C[A/a], q) = 0$ then, by the common property of all transformation functions, $a'.s(q) = tr_f(dv(s)) = dv(s)$. In the same time, $a''.s(q) = dv(s)$ by the definition of SELECT . Thus, $a'.s(q) = a''.s(q)$.

We have shown that for any arbitrary stream s and time quantum q, $a'.s(q) = a''.s(q)$. Therefore, $a' = a''$.

Theorem 3.3.4 (Swapping APPLY and BETWEEN) *Given an* APPLY *operator with a value-local transformation function tr_f and a* BETWEEN *operator with conditions*

C_1, C_2 *such that* $f \notin fp_A(C_1) \cup fp_A(C_2)$*, the following holds:*

$$\alpha(\beta_{C_1}^{C_2}(a), tr_f) \;=\; \beta_{C_1}^{C_2}(\alpha(a, tr_f)).$$

Proof of Theorem 3.3.4. This proof is similar to the proof of Theorem 3.3.3. As the theorem requires that $f \notin fp_A(C_1) \cap fp_A(C_2)$, we can say that $\lambda(C_1[A/a], q) = \lambda(C_1[A/\alpha(a, tr_f)], q)$ and $\lambda(C_2[A/a], q) = \lambda(C_2[A/\alpha(a, tr_f)], q)$ for any arbitrary quantum q. Due to the value-locality of tr_f, we can also say that each quantum can be considered individually, as it is not affected by the values at other quanta. Consider then an arbitrary stream s in $a' = \alpha(\beta_{C_1}^{C_2}(a), tr_f)$ and $a'' = \beta_{C_1}^{C_2}(\alpha(a, tr_f))$ at a time quantum q:

1. If $s \neq f$ then $a'.s(q) = \beta_{C_1}^{C_2}(a).s(q)$ and $a''.s(q) = \beta_{C_1}^{C_2}(a).s(q)$ by the definition of APPLY . Thus, $a'.s(q) = a''.s(q)$.

2. If $s = f$ and $\exists q' \leq q\ \lambda(C_1[A/a], q') = 1\ \wedge\ \forall q'' \in [q', q]\ \lambda(C_2[A/a], q'') = 0$ is true then, by the definition of APPLY and the theorem requirements, $a'.s(q) = tr_f(a.s(q))$ and does not depend on the values of $\beta_{C_1}^{C_2}(a)$ at any quanta other than q. At the same time, $a''.q(s) = tr_f(a.s(q))$, by the definition of BETWEEN . Thus, $a'.s(q) = a''.s(q)$.

3. If $s = f$ and $\exists q' \leq q\ \lambda(C_1[A/a], q') = 1\ \wedge\ \forall q'' \in [q', q]\ \lambda(C_2[A/a], q'') = 0$ is false then $a'.s(q) = tr_f(dv(s)) = dv(s)$ by the common property of all transformation functions. At the same time, $a''.s(q) = dv(s)$ by the definition of BETWEEN . Thus, $a'.s(q) = a''.s(q)$.

We have shown that for any arbitrary stream s and time quantum q, $a'.s(q) = a''.s(q)$. Therefore, $a' = a''$.

3.3.5 The PROJECT Operator

Similarly to the relational projection, the PROJECT operator is used to remove certain streams from an audio file while leaving the other streams unchanged. As in the case of selection, the removal is done by replacing data with default values from the schema.

The PROJECT operator is rarely used on its own, but often as a part of a larger query. For instance, when processing the right channel of a stereo audio recording, one may want to remove this channel from the original recording and add modified data later.

Definition 3.3.13 (PROJECT Operator) *Given an audio file a (database ADB) and a set of stream names $\mathcal{F}$, the* PROJECT operator *produces a new audio file (database)*

$$\begin{aligned} \pi_{\mathcal{F}}(a) &= \langle a.\ell, a.\delta t, \{f \mid f(q) = \begin{cases} dv(f) & if\ f \in \mathcal{F} \\ a.f(q) & otherwise \end{cases} \}\rangle, \\ \pi_{\mathcal{F}}(\mathsf{ADB}) &= \{\pi_{\mathcal{F}}(a) \mid a \in \mathsf{ADB}\}. \end{aligned}$$

Example 3.3.3 *Continuing with the stereo recording example, one can remove the right channel from an audio file a by issuing $\pi_{\{vol_r\}}(a)$ query. Alternatively, if we are only interested in the phoneme stream and text transcript of an audio file a, we can restrict a to these streams via the query $\pi_{\overline{\{ph,text\}}}(a)$.*

Several theorems apply to the reordering of PROJECT and other operators:

Theorem 3.3.5 (Swapping PROJECT Operators) *Given two* PROJECT *operators, the following holds:*

$$\pi_{\mathcal{F}_1}(\pi_{\mathcal{F}_2}(a)) = \pi_{\mathcal{F}_2}(\pi_{\mathcal{F}_1}(a)).$$

Proof of Theorem 3.3.5. Consider an arbitrary stream f in $a' = \pi_{\mathcal{F}_2}(\pi_{\mathcal{F}_1}(a))$ and $a'' = \pi_{\mathcal{F}_1}(\pi_{\mathcal{F}_2}(a))$ at a time quantum q:

1. If $f \notin \mathcal{F}_1$ and $f \notin \mathcal{F}_2$ then $a'.f(q) = a.f(q)$ and $a''.f(q) = a.f(q)$ by the definition of PROJECT . Thus, $a'.f(q) = a''.f(q)$.

2. If $f \in \mathcal{F}_1$ and $f \notin \mathcal{F}_2$ then $a.f(q)$ will be replaced with $dv(f)$ in one PROJECT , and left unchanged in the other. Thus, $a'.f(q) = dv(f)$, $a''.f(q) = dv(f)$, and therefore $a'.f(q) = a''.f(q)$.

3. If $f \in \mathcal{F}_2$ and $f \notin \mathcal{F}_1$, the situation is the same as in (2).

4. If $f \in \mathcal{F}_1$ and $f \in \mathcal{F}_2$ then $a.f(q)$ will be replaced with $dv(f)$ twice, in both PROJECT operators. Again, $a'.f(q) = dv(f)$, $a''.f(q) = dv(f)$, and therefore $a'.f(q) = a''.f(q)$.

We have shown that for any arbitrary stream f and time quantum q, $a'.f(q) = a''.f(q)$. Therefore, $a' = a''$.

Theorem 3.3.6 (Swapping PROJECT and SELECT) *Given a* PROJECT *operator that removes streams listed in $\mathcal{F}$ and a* SELECT *operator with condition C such that $\mathcal{F} \cap fp_A(C) = \emptyset$, the following holds:*

$$\pi_{\mathcal{F}}(\sigma_C(a)) \quad = \quad \sigma_C(\pi_{\mathcal{F}}(a)).$$

Proof of Theorem 3.3.6. Consider an arbitrary stream f in $a' = \pi_{\mathcal{F}}(\sigma_C(a))$ and $a'' = \sigma_C(\pi_{\mathcal{F}}(a))$ at a time quantum q:

1. If $f \notin \mathcal{F}$ and $\mathcal{F} \cap fp_A(C) = \emptyset$, as the theorem requires, then $a'.f(q) = \sigma_C(a).f(q)$ and $a''.f(q) = \sigma_C(a).f(q)$ by the definition of PROJECT . Thus, $a'.f(q) = a''.f(q)$.

2. If $f \in \mathcal{F}$ then $a'.f(q) = dv(f)$, $a''.f(q) = dv(f)$, and thus $a'.f(q) = a''.f(q)$.

We have shown that for any arbitrary stream f and time quantum q, $a'.f(q) = a''.f(q)$. Therefore, $a' = a''$.

Theorem 3.3.7 (Swapping PROJECT and BETWEEN) *Given a* PROJECT *operator that removes streams listed in $\mathcal{F}$ and a* BETWEEN *operator with conditions C_1, C_2 such that $\mathcal{F} \cap (fp_A(C_1) \cup fp_A(C_2)) = \emptyset$, the following holds:*

$$\pi_{\mathcal{F}}(\beta_{C_1}^{C_2}(a)) \quad = \quad \beta_{C_1}^{C_2}(\pi_{\mathcal{F}}(a)).$$

Proof of Theorem 3.3.7. This proof is similar to the proof of Theorem 3.3.6. Consider an arbitrary stream f in $a' = \pi_{\mathcal{F}}(\beta_{C_1}^{C_2}(a))$ and $a'' = \beta_{C_1}^{C_2}(\pi_{\mathcal{F}}(a))$ at a time quantum q:

1. If $f \notin \mathcal{F}$ and $\mathcal{F} \cap (fp_A(C_1) \cup fp_A(C_2)) = \emptyset$ (as the theorem requires) then $a'.f(q) = \beta_{C_1}^{C_2}(a).f(q)$ and $a''.f(q) = \beta_{C_1}^{C_2}(a).f(q)$ by the definition of PROJECT . Thus, $a'.f(q) = a''.f(q)$.

2. If $f \in \mathcal{F}$ then $a'.f(q) = dv(f)$, $a''.f(q) = dv(f)$, and thus $a'.f(q) = a''.f(q)$.

We have shown that for any arbitrary stream f and time quantum q, $a'.f(q) = a''.f(q)$. Therefore, $a' = a''$.

Theorem 3.3.8 (Swapping PROJECT and APPLY) *Given a* PROJECT *operator that removes streams listed in $\mathcal{F}$ and an* APPLY *operator with transformation function tr_f such that $\mathcal{F} \cap fp(tr_f) = \emptyset$ or $\mathcal{F} \cap fp(tr_f) = \{f\}$, the following holds:*

$$\pi_{\mathcal{F}}(\alpha(a, tr_f)) \quad = \quad \alpha(\pi_{\mathcal{F}}(a), tr_f).$$

Proof of Theorem 3.3.8. Consider an arbitrary stream s in $a' = \pi_{\mathcal{F}}(\alpha(a, tr_f))$ and $a'' = \alpha(\pi_{\mathcal{F}}(a), tr_f)$ at a time quantum q:

1. If $s \notin \mathcal{F}$ and $s \neq f$ then $a'.s(q) = a.s(q)$ and $a''.s(q) = a.s(q)$ by the definitions of PROJECT and APPLY operators. Thus, $a'.s(q) = a''.s(q)$.

2. If $s \notin \mathcal{F}$ and $s = f$ then the theorem requires that $\mathcal{F} \cap fp(tr_f) = \emptyset$ i.e. none of the deleted streams affect the value of tr_f. Thus $a'.s(q) = tr_f(a.s(q))$ and $a''.s(q) = tr_f(a.s(q))$ by the definitions of PROJECT and APPLY operators, and $a'.s(q) = a''.s(q)$.

3. If $s \in \mathcal{F}$ and $s = f$ then $a'.s(q) = dv(s)$ by the definition of PROJECT and $a''.s(q) = tr_f(dv(s)) = dv(s)$ by the common property of all transformation functions. Thus, $a'.s(q) = a''.s(q)$.

4. If $s \in \mathcal{F}$ and $s \neq f$ then $a'.s(q) = dv(s)$ and $a''.s(q) = dv(s)$ by the definitions of PROJECT and APPLY operators. Thus, $a'.s(q) = a''.s(q)$.

We have shown that for any arbitrary stream s and time quantum q, $a'.s(q) = a''.s(q)$. Therefore, $a' = a''$.

3.3.6 The MIX Operator

The *mix* of two audio files is a conditional merge of all streams constituting the files. This operator can be used to merge audio recordings from different sources. For example, when recording a vocal performance, the singer's voice and the instrumental sound track are usually picked up by different microphones and mixed in the "right" proportion by a person in charge of the audio. In the algebra, the same operation is done with the MIX operator.

However, before proceeding to define the MIX operator, some intermediate definitions are needed. In particular, we need to describe what it means to merge two values (of the same stream from two different audio files).

Definition 3.3.14 (Merging Policy) *A* merging policy *for a stream f is a function that takes as input two values $v_1, v_2 \in dom(f)$, and returns as output, a single value from $dom(f)$ such that $mp(v_1, dv(f)) = v_1$ and $mp(dv(f), v_2) = v_2$.*

It is worth noting that the default merging policy for a stream should be included into that stream's *schema*. Numerous merging policies exist. Some examples are given below.

Example 3.3.4 (Merging Policies) *One simple example of a merging policy would, given that $v_1 \neq dv(f)$ and $v_2 \neq dv(f)$, take an average $mp_{avg}(v_1, v_2) = (v_1 + v_2)/2$. In some cases, this policy may cause certain unwelcome effects though. For example, two quiet sounds will become even quieter after mixing their amplitudes with mp_{avg}. In such cases, one may use the cut-off policy $mp_{cut}(v_1, v_2) = min(f_{max}, |v_1 + v_2|) \cdot sgn(v_1 + v_2)$. Notice that if $dv(f) = 0$, mp_{cut} will not require any additional checking for it.*

We are now ready to define the MIX operator.

Definition 3.3.15 (MIX Operator) *Given two audio files a_1, a_2 (databases $\mathsf{ADB}_1, \mathsf{ADB}_2$) such that $a_1.\delta t = a_2.\delta t$, and a selection condition C with two free audio variables A_1, A_2, the* MIX operator *produces a new audio file (database) such that*

$$\begin{aligned} a_1 \otimes_C a_2 &= \langle max(a_1.\ell, a_2.\ell), a_1.\delta t, F \rangle, \\ \mathsf{ADB}_1 \otimes_C \mathsf{ADB}_2 &= \{a_1 \otimes_C a_2 \mid a_1 \in \mathsf{ADB}_1 \wedge a_2 \in \mathsf{ADB}_2\}, \end{aligned}$$

where

$$F = \{f \mid f(q) = \begin{cases} mp_f(a_1.f(q), a_2.f(q)) & if\ \lambda(C[A_1/a_1, A_2/a_2], q) = 1 \\ a_1.f(q) & otherwise \end{cases} \}.$$

It is assumed that $a.f(q) = dv(f)$ *for* $q \geq a.\ell$*. The* mp *policies used by the* MIX *operator are taken from the database schema and can be parameterized by the implementation, if needed.*

Notice that by definition given above, MIX is *not* symmetric, i.e. $a_1 \otimes_C a_2 \neq a_2 \otimes_C a_1$. When the MIX selection condition is $TRUE$, we will omit it and write $a_1 \otimes a_2$.

One important use of MIX is to merge results of other operators applied to the same audio file. It is often necessary to transform only those parts of a file that satisfy a certain condition. For example, one may want to muffle pieces of a musical composition where a tuba plays. To facilitate this kind of queries, one can use MIX for *conditional application* via the derived conditional APPLY operator.

Example 3.3.5 (Conditional APPLY Operator) *Given a transformation function* tr_f *and a selection condition* C *with a single free audio variable* A*, and an audio file* a *(database* ADB*), the* conditional APPLY operator *produces a new audio file (database)*

$$\begin{aligned} \alpha^C_{tr_f}(a) &= \alpha(\sigma_C(a), tr_f) \otimes \sigma_{\neg C}(a), \\ \alpha^C_{tr_f}(\mathsf{ADB}) &= \{\alpha^C_{tr_f}(a) \mid a \in \mathsf{ADB}\}. \end{aligned}$$

To continue with our example, given a musical recording a, one can muffle the tuba in it by issuing the $\alpha^{tuba \in A.midi}_{vol(q)=vol(q)/2}(a)$ query. There are several useful theorems showing how MIX can be combined with other operators.

Theorem 3.3.9 (Swapping MIX and APPLY) *Given an* APPLY *operator with a value-local transformation function* tr_f *and a* MIX *operator merging audio files on condition* C *such that* $f \notin fp_{A_1}(C) \cup fp_{A_2}(C)$ *and* $tr_f(mp(v_1, v_2)) = mp(tr_f(v_1), tr_f(v_2))$ *for all* $v_1, v_2 \in dom(f)$*, the following holds:*

$$\alpha(a_1 \otimes_C a_2, tr_f) = \alpha(a_1, tr_f) \otimes_C \alpha(a_2, tr_f).$$

Proof of Theorem 3.3.9. As the theorem requires that $f \notin fp_{A_1}(C) \cup fp_{A_2}C$, $\lambda(C[A_1/a_1, A_2/a_2], q)$ does not depend on the value of f. Also, as the theorem requires tr_f to be local, we can consider each quantum independently from all other quanta and its location in the stream. Consider then an arbitrary stream s in $a' = \alpha(a_1, tr_f) \otimes_C \alpha(a_2, tr_f)$ and $a'' = \alpha(a_1 \otimes_C a_2, tr_f)$ at a time quantum q:

1. If $s \neq f$ then by the definition of APPLY and theorem requirements, we can say that $a'.s(q) = a_1 \otimes_C a_2.s(q)$ and $a''.s(q) = a_1 \otimes_C a_2.s(q)$ i.e. $a'.s(q) = a''.s(q)$.

2. If $s = f$ and $\lambda(C[A_1/a_1, A_2/a_2], q) = 0$ then we have $a'.s(q) = tr_f(a_1.s(q))$ and $a''.s(q) = tr_f(a_1.s(q))$ i.e. $a'.s(q) = a''.s(q)$.

3. If $s = f$ and $\lambda(C[A_1/a_1, A_2/a_2], q) = 1$ then $a'.s(q) = mp(tr_f(a_1.s(q)), tr_f(a_2.s(q))$ and $a''.s(q) = tr_f(mp(a_1.s(q), a_2.s(q)))$ by the definitions of APPLY and MIX operators. As the theorem requires $tr_f(mp(v_1, v_2)) = mp(tr_f(v_1), tr_f(v_2))$, we again conclude that $a'.s(q) = a''.s(q)$.

We have shown that for any arbitrary stream s and time quantum q, $a'.s(q) = a''.s(q)$. Therefore, $a' = a''$.

Theorem 3.3.10 (Swapping MIX and PROJECT) *Given a* PROJECT *operator that removes streams listed in $\mathcal{F}$ and a* MIX *operator merging audio files on condition C such that $\mathcal{F} \cap fp_{A_1}(C) = \emptyset$ and $\mathcal{F} \cap fp_{A_2}(C) = \emptyset$, the following holds:*

$$\pi_{\mathcal{F}}(a_1 \otimes_C a_2) \quad = \quad \pi_{\mathcal{F}}(a_1) \otimes_C \pi_{\mathcal{F}}(a_2).$$

Proof of Theorem 3.3.10. Consider an arbitrary stream f in $a' = \pi_{\mathcal{F}}(a_1) \otimes_C \pi_{\mathcal{F}}(a_2)$ and $a'' = \pi_{\mathcal{F}}(a_1 \otimes_C a_2)$ at a time quantum q:

1. The theorem requires that $\mathcal{F} \cap fp_{A_1}(C) = \emptyset$ and $\mathcal{F} \cap fp_{A_2}(C) = \emptyset$ i.e. any deleted streams do not affect the value of $\lambda(C[A_1/a_1, A_2/a_2], q)$. Then if $f \notin \mathcal{F}$, by the PROJECT definition $a'.f(q) = a_1 \otimes_C a_2.f(q)$ and $a''.f(q) = a_1 \otimes_C a_2.f(q)$, and thus $a'.f(q) = a''.f(q)$.

2. If $f \in \mathcal{F}$ and $\lambda(C[A_1/a_1, A_2/a_2], q) = 0$ then we have $a'.f(q) = \pi_{\mathcal{F}}(a_1).f(q) = dv(f)$ and $a''.f(q) = \pi_{\mathcal{F}}(a_1).f(q) = dv(f)$ by the definition of the MIX operator, and therefore $a'.f(q) = a''.f(q)$.

3. If $f \in \mathcal{F}$ and $\lambda(C[A_1/a_1, A_2/a_2], q) = 1$ then $a'.f(q) = mp(dv(f), dv(f)) = dv(f)$ by the common property of all merging policies), and $a''.f(q) = dv(f)$ by the definition of PROJECT . Once again we conclude that $a'.f(q) = a''.f(q)$.

We have shown that for any arbitrary stream f and time quantum q, $a'.f(q) = a''.f(q)$. Therefore, $a' = a''$.

3.3.7 The Concatenation Operator

A common, albeit simple, audio processing task is the *concatenation* of audio files. Hence is the algebraic operator to do it.

Definition 3.3.16 (Concatenation Operator) *Given two audio files a_1, a_2 (databases $\mathsf{ADB}_1, \mathsf{ADB}_2$) such that $a_1.\delta t = a_2.\delta t$, the* concatenation operator *produces a new audio file (database)*

$$
\begin{aligned}
a_1 \oplus a_2 &= \langle a_1.\ell + a_2.\ell, a_1.\delta t, F \rangle, \\
\mathsf{ADB}_1 \oplus \mathsf{ADB}_2 &= \{a_1 \oplus a_2 \mid a_1 \in \mathsf{ADB}_1 \wedge a_2 \in \mathsf{ADB}_2\}, \\
where\ F &= \{f \mid f(q) = \begin{cases} a_1.f(q) & if\ q \in [0, a_1.\ell) \\ a_2.f(q - a_1.\ell) & otherwise \end{cases} \}.
\end{aligned}
$$

I now present some useful equivalences involving concatenation.

Theorem 3.3.11 (Swapping Concatenation Operators) *Given two concatenation operators, the following holds:*

$$(a_1 \oplus a_2) \oplus a_3 \quad = \quad a_1 \oplus (a_2 \oplus a_3).$$

Proof of Theorem 3.3.11. Consider an arbitrary stream f in $a' = (a_1 \oplus a_2) \oplus a_3$ and $a'' = a_1 \oplus (a_2 \oplus a_3)$ at a time quantum q. Due to the definition of the concatenation operator, we can write the following correspondence between stream values:

$$a'.f(q) \quad = \quad \begin{cases} a_1.f(q) & if\ q \in [0, a_1.\ell) \\ a_2.f(q - a_1.\ell) & if\ q \in [a_1.\ell, a_2.\ell) \\ a_3.f(q - a_1.\ell - a_2.\ell) & otherwise \end{cases}$$

$$a''.f(q) \quad = \quad \begin{cases} a_1.f(q) & if\ q \in [0, a_1.\ell) \\ a_2.f(q - a_1.\ell) & if\ q \in [a_1.\ell, a_2.\ell) \\ a_3.f(q - a_1.\ell - a_2.\ell) & otherwise \end{cases}$$

As $a'.f(q) = a''.f(q)$ for every stream f at any quantum q, one can conclude that a' and a'' are equivalent.

Theorem 3.3.12 (Swapping Concatenation and SELECT **)** *Given a* SELECT *operator with a local selection condition C, the following holds:*

$$\sigma_C(a_1 \oplus a_2) \quad = \quad \sigma_C(a_1) \oplus \sigma_C(a_2).$$

Proof of Theorem 3.3.12. Consider an arbitrary stream f in $a' = \sigma_C(a_1 \oplus a_2)$ and $a'' = \sigma_C(a_1) \oplus \sigma_C(a_2)$ at a time quantum q. Due to the definition of the concatenation operator and locality of C, we can write the following correspondence between stream

values:

$$a'.f(q) = \begin{cases} \sigma_C(a_1).f(q) & if\ q \in [0, a_1.\ell) \\ \sigma_C(a_2).f(q - a_1.\ell) & otherwise \end{cases}$$

$$a''.f(q) = \begin{cases} \sigma_C(a_1).f(q) & if\ q \in [0, a_1.\ell) \\ \sigma_C(a_2).f(q - a_1.\ell) & otherwise \end{cases}$$

As $a'.f(q) = a''.f(q)$ for every stream f at any quantum q, one can conclude that a' and a'' are equivalent.

Theorem 3.3.13 (Swapping Concatenation and APPLY **)** *Given an* APPLY *operator with a local transformation function tr_f and a concatenation of two audio files a_1, a_2, the following holds:*

$$\alpha(a_1 \oplus a_2, tr_f) = \alpha(a_1, tr_f) \oplus \alpha(a_2, tr_f).$$

Proof of Theorem 3.3.13. This proof is similar to the proof of Theorem 3.3.12. Consider an arbitrary stream f in $a' = \alpha(a_1 \oplus a_2, tr_f)$ and $a'' = \alpha(a_1, tr_f) \oplus \alpha(a_2, tr_f)$ at a time quantum q. Due to the definition of the concatenation operator and the locality of tr_f, we can write the following correspondence between stream values:

$$a'.f(q) = \begin{cases} \alpha(a_1, tr_f).f(q) & if\ q \in [0, a_1.\ell) \\ \alpha(a_2, tr_f).f(q - a_1.\ell) & otherwise \end{cases}$$

$$a''.f(q) = \begin{cases} \alpha(a_1, tr_f).f(q) & if\ q \in [0, a_1.\ell) \\ \alpha(a_2, tr_f).f(q - a_1.\ell) & otherwise \end{cases}$$

As $a'.f(q) = a''.f(q)$ for every stream f at any quantum q, one can conclude that a' and a'' are equivalent.

Theorem 3.3.14 (Swapping Concatenation and PROJECT) *Given a* PROJECT *operator that removes streams listed in $\mathcal{F}$ and a concatenation of two audio files a_1, a_2, the following holds:*

$$\pi_{\mathcal{F}}(a_1 \oplus a_2) \quad = \quad \pi_{\mathcal{F}}(a_1) \oplus \pi_{\mathcal{F}}(a_2).$$

Proof of Theorem 3.3.14. This proof is similar to the proof of Theorem 3.3.12. Consider an arbitrary stream f in $a' = \pi_{\mathcal{F}}(a_1 \oplus a_2)$ and $a'' = \pi_{\mathcal{F}}(a_1) \oplus \pi_{\mathcal{F}}(a_2)$ at a time quantum q. Due to the definition of the concatenation operator, we can write the following correspondence between stream values:

$$a'.f(q) \quad = \quad \begin{cases} \pi_{\mathcal{F}}(a_1).f(q) & if\ q \in [0, a_1.\ell) \\ \pi_{\mathcal{F}}(a_2).f(q - a_1.\ell) & otherwise \end{cases}$$

$$a''.f(q) \quad = \quad \begin{cases} \pi_{\mathcal{F}}(a_1).f(q) & if\ q \in [0, a_1.\ell) \\ \pi_{\mathcal{F}}(a_2).f(q - a_1.\ell) & otherwise \end{cases}$$

As $a'.f(q) = a''.f(q)$ for every stream f at any quantum q, one can conclude that a' and a'' are equivalent.

3.3.8 The RESAMPLE Operator

Recall that the inputs to such binary operators as MIX and concatenation are required to have the same δt. However, it is often the case that we wish to perform these operations (e.g. mixing) on two audio files sampled at different frequencies and thus having different δt values. This can be achieved by "resampling" one of the files (usually the one with the larger δt, to preserve fidelity). Resampling has many other uses, too. For example, when recording an audio CD, it is important to resample all audio data to 44.1kHz required by the AudioCD standard. On the other hand, when playing audio through a phone line, or storing it on a limited-capability device (such

as a cellular phone) one would have to resample it to the 8kHz frequency. To facilitate resampling, let us first introduce the *interpolation policy*:

Definition 3.3.17 (Stream Interpolation Policy) *Given a stream f, time t, and a period δt, the* stream interpolation policy $ip(f, t, \delta t)$ *returns a value from* $dom(f)$ *such that* $ip(f, i \cdot \delta t, \delta t) = f(i)$.

It is worth noting that the default interpolation policy for a stream should be included into that stream's *schema*.

Let us examine what this definition says w.r.t. Figure 3.1. Suppose $f.v = \langle\ 0.1,\ 0.3,\ 0.4,\ 0.5,\ 0.4,\ 0.3,\ 0.2,\ 0.1,\ 0\ \rangle$. Consider any policy ip for this case. Here, $\delta t = 2\mu s$. $ip(f, 7, 0.000002)$ can be any number whatsoever between 0 and 1. However, $ip(f, 6, 2)$ must equal 0.5 because $6 = 3 \cdot \delta t$ and $f(3) = 0.5$. Examples of some interpolation policies are given below.

Example 3.3.6 (Interpolation Policies) *The simplest interpolation policy would just return the previous or the next known value of f:*

$$\begin{aligned} ip_{prev}(f, t, \delta t) &= f(\lfloor t/\delta t \rfloor) \\ ip_{next}(f, t, \delta t) &= f(\lfloor t/\delta t \rfloor + 1) \end{aligned}$$

If there is a complete ordering on $dom(f)$, one may want to take the smallest or the largest neighboring value:

$$\begin{aligned} ip_{min}(f, t, \delta t) &= min(f(\lfloor t/\delta t \rfloor), f(\lfloor t/\delta t \rfloor + 1)) \\ ip_{max}(f, t, \delta t) &= max(f(\lfloor t/\delta t \rfloor), f(\lfloor t/\delta t \rfloor + 1)) \end{aligned}$$

Finally, if f is numeric, it is possible to interpolate it linearly:

$$ip_{lin}(f, t, \delta t) = f(\lfloor t/\delta t \rfloor) + \frac{(f(\lfloor t/\delta t \rfloor + 1) - f(\lfloor t/\delta t \rfloor)(t - \lfloor t/\delta t \rfloor \delta t)}{\delta t}$$

More complicated interpolation policies may involve quadratic or spline interpolation. An interpolation policy may not be applicable to some data types. For instance, ip_{lin} can't be applied to set, character, and string domains.

Armed with stream interpolation policies, we can now introduce the RESAMPLE operator:

Definition 3.3.18 (RESAMPLE Operator) *Given an audio file a (database ADB) and a time period $\delta t'$, the* RESAMPLE operator *returns a new audio file (database) such that*

$$\begin{aligned} \gamma_{\delta t'}(a) &= \langle \lfloor \frac{a.\ell \cdot a.\delta t}{\delta t'} \rfloor, \delta t', \{f \mid f(q) = ip(a.f, q \cdot \delta t', a.\delta t)\} \rangle, \\ \gamma_{\delta t'}(\mathsf{ADB}) &= \{\gamma_{\delta t'}(a) \mid a \in \mathsf{ADB}\}. \end{aligned}$$

The ip policies used by the RESAMPLE *operator are taken from the database schema and can be parameterized by the implementation, if needed.*

Under certain conditions, RESAMPLE can be swapped with SELECT , PROJECT , and APPLY :

Theorem 3.3.15 (Swapping RESAMPLE and SELECT) *Suppose C is a local selection condition and either ip_{next} or ip_{prev} is the stream interpolation policy used for all streams in the audio schema. Then:*

$$\gamma_{\delta t'}(\sigma_C(a)) = \sigma_C(\gamma_{\delta t'}(a)).$$

Proof of Theorem 3.3.15. Assume that interpolation policy does not create any new values, but provides a correspondence $w : \Re \rightarrow [0, \ell)$ between time instances and audio file quanta. Both ip_{prev} and ip_{next} are such policies. Let us then consider an arbitrary stream f in $a' = \gamma_{\delta t'}(\sigma_C(a))$ and $a'' = \sigma_C(\gamma_{\delta t'}(a))$ at a time quantum

q. As the theorem requires C to be local, we can look at each quantum separately. Both $a'.f(q)$ and $a''.f(q)$ will return the result of $\sigma_C(a)$ operator at $w(q \cdot \delta t')$. Thus, $a'.f(q) = a''.f(q)$ and because it is true for any arbitrary stream f at arbitrary moment q, $a' = a''$.

Theorem 3.3.16 (Swapping RESAMPLE **and** PROJECT **)** *Suppose $\mathcal{F}$ is a list of streams to be removed from an audio file and either ip_{next} or ip_{prev} is the stream interpolation policy used for all streams in the audio schema. Then:*

$$\gamma_{\delta t'}(\pi_{\mathcal{F}}(a)) \quad = \quad \pi_{\mathcal{F}}(\gamma_{\delta t'}(a)).$$

Proof of Theorem 3.3.16. This proof is similar to the proof of Theorem 3.3.15. Assume that interpolation policy does not create any new values, but provides a correspondence $w : \Re \rightarrow [0, \ell)$ between time instances and audio file quanta. Both ip_{prev} and ip_{next} are such policies. Let us then consider an arbitrary stream f in $a' = \gamma_{\delta t'}(\pi_{\mathcal{F}}(a))$ and $a'' = \pi_{\mathcal{F}}(\gamma_{\delta t'}(a))$ at a time quantum q. Both $a'.f(q)$ and $a''.f(q)$ will return the result of $\pi_{\mathcal{F}}(a)$ operator at $w(q \cdot \delta t')$. Thus, $a'.f(q) = a''.f(q)$ and because it is true for any arbitrary stream f at arbitrary moment q, $a' = a''$.

Theorem 3.3.17 (Swapping RESAMPLE **and** APPLY **)** *Suppose tr_f is a local stream transformation function and either ip_{next} or ip_{prev} is the stream interpolation policy used for all streams in the audio schema. Then:*

$$\gamma_{\delta t'}(\alpha(a, tr_f)) \quad = \quad \alpha(\gamma_{\delta t'}(a), tr_f).$$

Proof of Theorem 3.3.17. This proof is similar to the proof of Theorem 3.3.15. Assume that interpolation policy does not create any new values, but provides a correspondence $w : \Re \rightarrow [0, \ell)$ between time instances and audio file quanta. Both ip_{prev} and ip_{next} are such policies. Also, the theorem requires tr_f to be local i.e. we can

look at each time quantum individually, without taking into account its location or other quanta. Let us then consider an arbitrary stream f in $a' = \gamma_{\delta t'}(\alpha(a, tr_f))$ and $a'' = \alpha(\gamma_{\delta t'}(a), tr_f)$ at time quantum q. Both $a'.f(q)$ and $a''.f(q)$ will return the result of $\alpha(a, tr_f)$ operator at $w(q \cdot \delta t')$ Thus, $a'.f(q) = a''.f(q)$ and because it is true for any arbitrary stream f at arbitrary moment q, $a' = a''$.

3.3.9 The COMPRESS Operator

In practice, one would often encounter audio files that contain "gaps" where default stream values are repeated for long periods of time. A typical example of such a file would be the output of a surveillance system or a recording from an aircraft's "black box" recorder with long periods of silence. By its nature, the SELECT operator also yields audio files that contain gaps. One may often want to compress such files by removing the gaps. This can be done with the COMPRESS operator, defined as follows:

Definition 3.3.19 (COMPRESS Operator) *Given an audio file a (database* ADB*) and a set of stream names $\mathcal{F}$, the* COMPRESS operator *returns a new audio file (database) such that*

$$\begin{aligned}
w_0 &= 0, \\
w_{i+1} &= q \; such \; that \; q \in (w_i, a.\ell) \; \wedge \; \exists f \in \mathcal{F} \; a.f(q) \neq dv(f) \; \wedge \\
&\quad \forall q' \in (w_i, q) \forall f \in \mathcal{F} \; a.f(q') = dv(f), \\
\ell &= i \; such \; that \; \forall w_j \; j \leq i, \\
\eta_{\mathcal{F}}(a) &= \langle \ell, a.\delta t, \{f \mid f(q) = a.f(w_{q+1})\} \rangle, \\
\eta_{\mathcal{F}}(\mathsf{ADB}) &= \{\eta_{\mathcal{F}}(a) \mid a \in \mathsf{ADB}\}.
\end{aligned}$$

Two COMPRESS operators can be swapped.

Theorem 3.3.18 (Swapping COMPRESS Operators) *Suppose $\mathcal{F}_1$, $\mathcal{F}_2$ are sets of streams and a is an audio file. Then:*

$$\eta_{\mathcal{F}_1}(\eta_{\mathcal{F}_2}(a)) \quad = \quad \eta_{\mathcal{F}_2}(\eta_{\mathcal{F}_1}(a)).$$

Proof of Theorem 3.3.18. Consider an arbitrary quantum q in an audio file a and two queries: $a' = \eta_{\mathcal{F}_2}(\eta_{\mathcal{F}_1}(a))$ and $a'' = \eta_{\mathcal{F}_1}(\eta_{\mathcal{F}_2}(a))$. By the definition of the COMPRESS operator, quantum q may have counterparts q' and q'' in a' and a'', such that for any audio stream f, $a'(q') = a(q)$ and $a''(q'') = a(q)$. It is also true that given two quanta $q_1 \leq q_2$ in a, their counterparts in a', a'' will have the same ordering, i.e. $q'_1 \leq q'_2$ and $q''_1 \leq q''_2$. Thus, to prove the equality $a' = a''$, we need to show that q will either be present in both a', a'' or absent from both of them. Now, consider a set of audio streams $\mathcal{F}$ such that $\forall f \in \mathcal{F}\ a.f(q) \neq dv(f)$ and $\forall f' \notin \mathcal{F}\ a.f'(q) = dv(f')$:

1. If $\mathcal{F} \cap \mathcal{F}_1 = \emptyset$ and $\mathcal{F} \cap \mathcal{F}_2 = \emptyset$ then the quantum q is not present in both $\eta_{\mathcal{F}_1}(a)$ and $\eta_{\mathcal{F}_2}(a)$. Therefore, it is absent from both a' and a''.

2. If $\mathcal{F} \cap \mathcal{F}_1 \neq \emptyset$ and $\mathcal{F} \cap \mathcal{F}_2 = \emptyset$ then the quantum q is included into $\eta_{\mathcal{F}_1}(a)$, but not into a', because of the outer $\eta_{\mathcal{F}_2}()$ operator. It is also not included into $\eta_{\mathcal{F}_2}(a)$ and therefore absent from a''.

3. If $\mathcal{F} \cap \mathcal{F}_1 = \emptyset$ and $\mathcal{F} \cap \mathcal{F}_2 \neq \emptyset$ then the quantum q is included into $\eta_{\mathcal{F}_2}(a)$, but not into a'', because of the outer $\eta_{\mathcal{F}_1}()$ operator. It is also not included into $\eta_{\mathcal{F}_1}(a)$ and therefore absent from a'.

4. Finally, if $\mathcal{F} \cap \mathcal{F}_1 \neq \emptyset$ and $\mathcal{F} \cap \mathcal{F}_2 \neq \emptyset$ then the quantum q is included into both a' and a''.

Thus, we have shown that any arbitrary quantum q in the input audio file a is either present in both a' and a'' or absent from both of them and the original ordering of quanta is preserved in both a' and a''. Therefore, $a' = a''$.

Under certain conditions, COMPRESS can be swapped with PROJECT , APPLY , and concatenation operators. COMPRESS can be swapped with PROJECT as long as the set of streams being compressed does not intersect with the projected streams.

Theorem 3.3.19 (Swapping COMPRESS and PROJECT) *Suppose $\mathcal{F}_1$, $\mathcal{F}_2$ are disjoint sets of streams and a is an audio file. Then:*

$$\eta_{\mathcal{F}_1}(\pi_{\mathcal{F}_2}(a)) \quad = \quad \pi_{\mathcal{F}_2}(\eta_{\mathcal{F}_1}(a)).$$

Proof of Theorem 3.3.19. The COMPRESS operator defines a $w : [0, \eta_{\mathcal{F}_1}(a).\ell) \rightarrow [0, a.\ell)$ mapping between time quanta in the compression result $\eta_{\mathcal{F}_1}(a)$ and the quanta in the input audio file a. As the theorem requires $\mathcal{F}_1 \cap \mathcal{F}_2 = \emptyset$, deletion of a stream $f \in \mathcal{F}_2$ will not affect w. Thus, let us consider an arbitrary stream f in $a' = \eta_{\mathcal{F}_2}(\pi_{\mathcal{F}_1}(a))$ and $a'' = \pi_{\mathcal{F}_1}(\eta_{\mathcal{F}_2}(a))$ at a time quantum q:

1. If $f \notin \mathcal{F}_2$ then, by the definition of PROJECT , $a'.f(q) = a.f(w(q))$ and $a''.f(q) = a.f(w(q))$, i.e. $a'.f(q) = a''.f(q)$.

2. If $f \in \mathcal{F}_2$ then, by the definition of PROJECT , $a'.f(q) = dv(f)$ and $a''.f(q) = dv(f)$, and again $a'.f(q) = a''.f(q)$.

As $a'.f(q) = a''.f(q)$ for every stream f at any quantum q, one can conclude that a' and a'' are equivalent.

COMPRESS can be swapped with APPLY as long as the stream affected by the APPLY operator is not a stream being compressed.

Theorem 3.3.20 (Swapping COMPRESS and APPLY) *Given a* COMPRESS *operator based on streams listed in $\mathcal{F}$ and an* APPLY *operator with a local transformation function tr_f, such that $f \notin \mathcal{F}$ or $x \neq dv(f) \rightarrow tr_f(x) \neq dv(f)$, the following holds for any audio file a:*

$$\eta_{\mathcal{F}}(\alpha(a, tr_f)) \quad = \quad \alpha(\eta_{\mathcal{F}}(a), tr_f).$$

Proof of Theorem 3.3.20. As the theorem requires tr_f to be local, the APPLY operates on each time quantum individually, we can consider each quantum individually, without taking into account its location or the data at the other quanta. The COMPRESS operator defines a $w : [0, \eta_{\mathcal{F}}(a).\ell) \rightarrow [0, a.\ell)$ mapping between time quanta in the compression result $\eta_{\mathcal{F}}(a)$ and the quanta in the input audio file a. Let us then consider an arbitrary stream s in $a' = \eta_{\mathcal{F}}(\alpha(a, tr_f))$ and $a'' = \alpha(\eta_{\mathcal{F}}(a), tr_f)$ at a time quantum q:

1. If $f \notin \mathcal{F}$ then for any stream $f' \in \mathcal{F}$ it is true that $\alpha(a, tr_f).f'(q) = a.f'(q)$, by the definition of APPLY . Then, by the definition of COMPRESS , the COMPRESS operators in both a' and a'' induce the same mapping w on their inputs. Therefore, $a'.s(q) = \alpha(a, tr_f).s(w(q))$ and $a''.s(q) = \alpha(a, tr_f).s(w(q))$, i.e. $a'.s(q) = a''.s(q)$.

2. If $f \in \mathcal{F}$ then the theorem requires that $x \neq dv(f) \rightarrow tr_f(x) \neq dv(f)$. Additionally, by the common property of all transformation functions, $x = dv(f) \rightarrow tr_f(x) = dv(f)$, i.e. $x = dv(f)$ if and only if $tr_f(x) = dv(f)$. By definition, APPLY can only modify a single stream f, therefore for any stream $f' \in \mathcal{F}$ it is true that $a.f'(q) = dv(f')$ if and only if $a.f'(q) = \alpha(a, tr_f).f'(q) = dv(f')$. Then, by the definition of COMPRESS , the COMPRESS operators in both a'

and a'' induce the same mapping w on their inputs. Therefore, $a'.s(q) = \alpha(a, tr_f).s(w(q))$ and $a''.s(q) = \alpha(a, tr_f).s(w(q))$, i.e. $a'.s(q) = a''.s(q)$.

As $a'.s(q) = a''.s(q)$ for every stream s at any quantum q, one can conclude that a' and a'' are equivalent.

Finally, COMPRESS can be swapped with the concatenation operator.

Theorem 3.3.21 (Swapping COMPRESS and Concatenation) *Given a* COMPRESS *operator and a concatenation of two audio files a_1, a_2, the following holds:*

$$\eta_{\mathcal{F}}(a_1 \oplus a_2) \;=\; \eta_{\mathcal{F}}(a_1) \oplus \eta_{\mathcal{F}}(a_1).$$

Proof of Theorem 3.3.21. This proof is similar to the proof of Theorem 3.3.12. Consider an arbitrary stream f in $a' = \eta_{\mathcal{F}}(a_1 \oplus a_2)$ and $a'' = \eta_{\mathcal{F}}(a_1) \oplus \eta_{\mathcal{F}}(a_2)$ at a time quantum q. By the definition of COMPRESS , there are two mappings $w_1 : [0, \eta_{\mathcal{F}}(a_1).\ell) \rightarrow [0, a_1.\ell)$ and $w_2 : [0, \eta_{\mathcal{F}}(a_2).\ell) \rightarrow [0, a_2.\ell)$ that determine which quanta are selected by the COMPRESS operator. Due to the definition of the concatenation operator, we can write the following correspondence between stream values:

$$a'.f(q) \;=\; \begin{cases} a_1.f(w_1(q)) & if\ q \in [0, a_1.\ell) \\ a_2.f(w_2(q - a_1.\ell)) & otherwise \end{cases}$$

$$a''.f(q) \;=\; \begin{cases} a_1.f(w_1(q)) & if\ q \in [0, a_1.\ell) \\ a_2.f(w_2(q - a_1.\ell)) & otherwise \end{cases}$$

As $a'.f(q) = a''.f(q)$ for every stream f at any quantum q, one can conclude that a' and a'' are equivalent.

3.3.10 The MATCH Operator

There are many cases where we have an audio file a and another audio file a' and want to find places in a that are most similar to a', or "match" a'. For instance, returning to our court recording, one may wish to find the k best matches for judge's statements where judge has spoken a certain word. Similar queries can often be useful for searching surveillance logs, transcripts, musical tunes, and so forth.

In order to support matching, we first need to define what it means for an audio stream to be similar to another audio stream.

Definition 3.3.20 (Distance Measure) *A distance measure dm is a mapping from pairs of audio streams to $[0, 1]$ such that $dm(f, f) = 0$ for any audio stream f.*

As f and f' become less similar, the value of $dm(f, f')$ grows. The default distance measure for two streams following the same schema should be included into that *schema*. Some specific distance measures are given below.

Example 3.3.7 (Distance Measures)

1. **Distance for numeric streams:** *Suppose streams f and f' are numeric and $\ell = min(length(f), length(f'))$. We may use the linear distance, the quadratic distance or the maximal distance to measure similarity between the streams.*

$$\begin{aligned} dm_{ld}(f, f') &= \frac{\sum_{i=0}^{\ell-1} |f(i) - f'(i)|}{\ell \cdot |max(f) - min(f)|}, \\ dm_{qd}(f, f') &= \frac{\sum_{i=0}^{\ell-1} (f(i) - f'(i))^2}{\ell \cdot (max(f) - min(f))^2}, \\ dm_{md}(f, f') &= \frac{max_{i=0}^{\ell-1} |f(i) - f'(i)|}{|max(f) - min(f)|}. \end{aligned}$$

   It follows from the definitions of these distance measures that for all f and f', $dm_{qd}(f, f') \leq dm_{ld}(f, f') \leq dm_{md}(f, f')$.

2. **Distance for set-valued streams:** *Suppose f and f' are set-valued streams, such as frequency spectrum or a set of currently playing musical instruments. We can apply the following distance measure to these streams:*

$$dm_{sd}(f, f') \quad = \quad \frac{1}{n} \sum_{i=0}^{\ell-1} \frac{card(f(i) \cap f'(i))}{card(f(i) \cup f'(i)) + 1}.$$

Other stream types may require different measures such as shortest editing distance for strings or perceived audio similarity for phonemes. Notice that sizes of two streams do not have to be equal. In fact, chunks of streams considered by dm may differ in size, although it is natural that the pattern f' is considered in its entirety.

Definition 3.3.21 (Multiple Stream Distance) *The distance between two audio files a, a' with respect to a set of stream names $\mathcal{F}$ can be computed as follows:*

$$dm_{\mathcal{F}}(a, a') \quad = \quad \sum_{f \in \mathcal{F}} dm(a.f, a'.f).$$

It is assumed that a and a' follow the same schema from which appropriate stream distance measures are taken.

Definition 3.3.22 (Match Ordering) *Suppose a, a' are audio files and $\mathcal{F}$ is a set of stream names. Let $SEQ(a)$ denote the set of all contiguous subsequences of a. Each $x \in SEQ(a)$ is a range of $[l, u]$ such that $l \in [0, a.\ell)$ and $u \in [l, a.\ell)$, and can be treated as an audio file by itself. Suppose $x_1, x_2 \in SEQ(a)$. We say that $x_1 \sqsubseteq_{a'} x_2$ iff $dm_{\mathcal{F}}(x_1, a') \leq dm_{\mathcal{F}}(x_2, a')$ and call $\sqsubseteq_{a'}$ the* match ordering *of $SEQ(a)$ w.r.t. a'.*

Now, the matching operator can be defined as follows.

Definition 3.3.23 (MATCH Operator) *Suppose a, a' are two audio files,* ADB *is an audio database, $\mathcal{F}$ is a set of stream names, k is a natural number, and $d_{max} \geq 0$*

is a real number. Let S consist of k smallest members of $SEQ(a)$ w.r.t. $\sqsubseteq_{a'}$ such that $\forall x \in S\ dm_{\mathcal{F}}(x, a') \leq d_{max}$. Then the MATCH operator *returns a new audio file (database) such that*

$$\mu^k_{\mathcal{F}}(a, a', d_{max}) = \langle a.\ell, a.\delta t, \{f \mid f(q) = \begin{cases} a.f(q) & if\ \exists x \in S\ q \in x \\ dv(f) & otherwise \end{cases} \}\rangle,$$

$$\mu^k_{\mathcal{F}}(\mathsf{ADB}, a', d_{max}) = \{x = \mu^k_{\mathcal{F}}(a, a') \mid a \in \mathsf{ADB} \wedge \exists f \exists q\ x.f(q) \neq dv(f)\}.$$

3.4 Optimizing SELECT and MIX

Two basic operators most often executed on audio data are SELECT and MIX . Given an audio file (database) and a selection condition, SELECT searches for quanta that satisfy this condition. A more complex operator, MIX , searches two audio files at once for quanta satisfying a joint selection condition.

In a very naive implementation, both selection and mixing can be performed by scanning input audio files, quantum by quantum, and writing out the resulting quanta as they are being computed. Due to the usual smoothness of audio data, its traversal can be accelerated by compressing audio streams with the *run-length encoding* (RLE). To further accelerate search operations though, one has to index audio streams and audio files. In this section, I will show how *numeric* audio streams (such as waveform or amplitude) can be indexed and searched for the purpose of selection and mixing.

To accelerate selection and mixing, one needs an efficient way to decide whether a range of quanta in an audio file contains any useful data with respect to a selection condition, and if it does not, skip over this range. For example, if we are looking at a range of quanta $[3 \cdot 10^6, 7 \cdot 10^6)$, it would be nice if we could quickly say that *nothing* in this range satisfies our selection condition, thus allowing us to jump over a large

range of quanta in our search. This operation can be abstracted with the following two API calls:

1. The $q' = f.SkipNIL(q)$ call skips stream f to the nearest quantum $q' \geq q$ such that $f(q') \neq dv(f)$. Thus, $SkipNIL()$ allows jumping over "empty" ranges of quanta.

2. The $q' = f.SkipTo(q, d_{min}, d_{max}, E)$ call skips stream f to the nearest quantum $q' \geq q$ such that $d_{min} \leq f(q') \leq d_{max}$ when $E = false$, or $f(q') < d_{min} \wedge f(q') > d_{max}$ when $E = true$. Thus, $SkipTo()$ allows jumping directly to stream ranges containing "interesting" data.

Let us start by dividing a stream into a sequence of segments, as shown in the Figure 3.4. Each segment $s = \langle start, end, min, max \rangle$ is characterized by the starting and ending quanta, minimal, and maximal values. A sequence of such segments shown in the Figure 3.4 is formally known as a *subdivision*. It is clear that a stream can be subdivided in multiple ways that are not equally good. Ideally, we would like each segment to be homogeneous and maximize the difference between each segment and its neighbors.

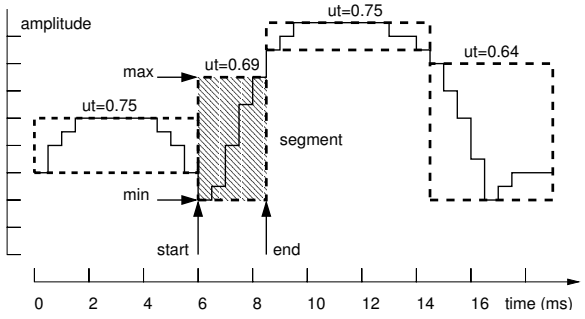


Figure 3.4: Subdivision.

To come up with a good segmentation criterion for efficient indexing, let us define the *utilization* of a segment to be

$$ut(s) \quad = \quad 1 - \frac{2 \cdot \sum_{i \in [s.start, s.end)} min(s.max - s(i), s(i) - s.min)}{(s.end - s.start) \cdot (s.max - s.min)}.$$

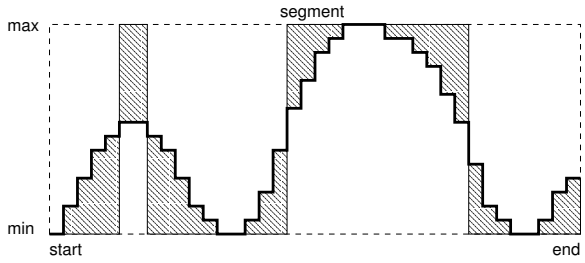


Figure 3.5: Utilization.

Figure 3.4 shows a subdivision of segments with their corresponding ut values, while Figure 3.5 shows a single segment. The ut function is directly proportional to the segment length and inversely proportional to the shaded area shown in Figure 3.5. Intuitively, higher values of ut correspond to segments that have less empty space above and below the waveform and thus better approximate the data. By setting a lower bound on ut, one can break an audio stream into segments of different coarseness with the following greedy algorithm:

```
Algorithm SegmentStream(Data,ut_min)
    Data is an audio stream
    ut_min ∈ [0,1] is a lower limit on utilization
begin
    Result := ∅
    Start := 0
    Max := Data(0)
    Min := Data(0)
    for j ∈ [1, length(Data)) do
        // If upper boundary has changed...
        if Data(j) > Max then
            // When ut falls below ut_min, create a segment
            if ut(⟨Start, j, Min, Data(j)⟩) < ut_min then
                Result := Result ∪ {⟨Start, j, Min, Max⟩}
                Start := j
                Min := Data(j)
```

```
            end if
            Max := Data(j)
        end if
        // If lower boundary has changed...
        if Data(j) < Min then
            // When ut falls below ut_min, create a segment
            if ut(⟨Start, j, Data(j), Max⟩) < ut_min then
                Result := Result ∪ {⟨Start, j, Min, Max⟩}
                Start := j
                Max := Data(j)
            end if
            Min := Data(j)
        end if
    end for
    Result := Result ∪ {⟨Start, length(Data), Min, Max⟩}
    return Result
end
```

The $SegmentStream()$ algorithm scans the stream and computes a new utilization value every time its upper or lower boundary changes. A new segment is created every time the ut value falls below the given lower bound ut_{min}. This ensures that all segments created by the algorithm have at least ut_{min} utilization, as shown in the following example.

Example 3.4.1 (Building a Subdivision) *Columns 3-4 of the Table 3.1 show steps taken by the $SegmentStream()$ algorithm called with $ut_{min} = 0.37$ on a stream shown in the Figure 3.4. Column 3 contains the currently scanned segment's length and boundaries, while column 4 shows the utilization. As one can see, the ut value falls below ut_{min} at the quantum 17, causing the algorithm to create a segment $\langle 0, 17, 4, 13\rangle$ and start on a new segment. In this example, the $SegmentStream()$ algorithm creates a subdivision of two segments: $\langle 0, 17, 4, 13\rangle$ and $\langle 17, 38, 4, 17\rangle$. The starting quanta of these two segments are marked with asterisks in the table.*

When creating a subdivision, one faces a tradeoff between the segment boundary tightness and the number of segments. To have *both* succinct and precise representations of an audio stream, one can use the *subdivision tree* data structure whose

		$ut \geq 0.37$		$ut \geq 0.74$	
q	f(q)	$\ell \times [min, max]$	ut	$\ell \times [min, max]$	ut
0	6	*1 $\times$ [6, 6]	1.00	*1 $\times$ [6, 6]	1.00
1	8	2 $\times$ [6, 8]	1.00	2 $\times$ [6, 8]	1.00
2	9	3 $\times$ [6, 9]	0.78	3 $\times$ [6, 9]	0.78
3	10	4 $\times$ [6, 10]	0.63	*1 $\times$ [10, 10]	1.00
4	10	5 $\times$ [6, 10]	0.70	2 $\times$ [10, 10]	1.00
5	10	6 $\times$ [6, 10]	0.75	3 $\times$ [10, 10]	1.00
6	10	7 $\times$ [6, 10]	0.79	4 $\times$ [10, 10]	1.00
7	10	8 $\times$ [6, 10]	0.81	5 $\times$ [10, 10]	1.00
8	10	9 $\times$ [6, 10]	0.83	6 $\times$ [10, 10]	1.00
9	9	10 $\times$ [6, 10]	0.80	7 $\times$ [9, 10]	1.00
10	8	11 $\times$ [6, 10]	0.73	8 $\times$ [8, 10]	0.88
11	6	12 $\times$ [6, 10]	0.75	9 $\times$ [6, 10]	0.83
12	4	13 $\times$ [4, 10]	0.74	10 $\times$ [4, 10]	0.83
13	5	14 $\times$ [4, 10]	0.74	11 $\times$ [4, 10]	0.82
14	8	15 $\times$ [4, 10]	0.71	12 $\times$ [4, 10]	0.78
15	11	16 $\times$ [4, 11]	0.57	*1 $\times$ [11, 11]	1.00
16	13	17 $\times$ [4, 13]	0.41	2 $\times$ [11, 13]	1.00
17	15	*1 $\times$ [15, 15]	1.00	*1 $\times$ [15, 15]	1.00
18	16	2 $\times$ [15, 16]	1.00	2 $\times$ [15, 16]	1.00
19	17	5 $\times$ [11, 17]	0.67	*1 $\times$ [17, 17]	1.00
20	17	6 $\times$ [11, 17]	0.72	2 $\times$ [17, 17]	1.00
21	17	7 $\times$ [11, 17]	0.76	3 $\times$ [17, 17]	1.00
22	17	8 $\times$ [11, 17]	0.79	4 $\times$ [17, 17]	1.00
23	17	9 $\times$ [11, 17]	0.82	5 $\times$ [17, 17]	1.00
24	17	10 $\times$ [11, 17]	0.83	6 $\times$ [17, 17]	1.00
25	17	11 $\times$ [11, 17]	0.85	7 $\times$ [17, 17]	1.00
26	16	12 $\times$ [11, 17]	0.83	8 $\times$ [16, 17]	1.00
27	16	13 $\times$ [11, 17]	0.82	9 $\times$ [16, 17]	1.00
28	15	14 $\times$ [11, 17]	0.79	10 $\times$ [15, 17]	0.80
29	14	15 $\times$ [11, 17]	0.73	11 $\times$ [14, 17]	0.82
30	12	16 $\times$ [11, 17]	0.73	12 $\times$ [12, 17]	0.80
31	10	17 $\times$ [10, 17]	0.73	13 $\times$ [10, 17]	0.80
32	7	18 $\times$ [7, 17]	0.71	14 $\times$ [7, 17]	0.79
33	4	19 $\times$ [4, 17]	0.73	15 $\times$ [4, 17]	0.79
34	5	20 $\times$ [4, 17]	0.73	16 $\times$ [4, 17]	0.79
35	6	21 $\times$ [4, 17]	0.73	17 $\times$ [4, 17]	0.78
36	6	22 $\times$ [4, 17]	0.73	18 $\times$ [4, 17]	0.78
37	6	23 $\times$ [4, 17]	0.73	19 $\times$ [4, 17]	0.77

Table 3.1: $SegmentStream()$ Operation.

branches correspond to the same audio stream segmented with different degrees of resolution, as shown in Figure 3.6.

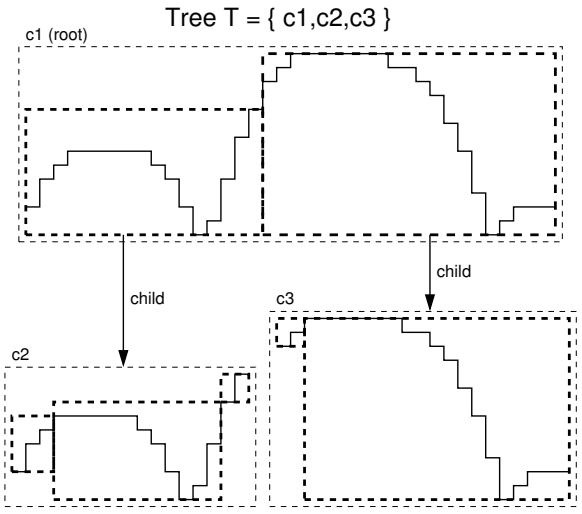


Figure 3.6: Subdivision Tree.

Definition 3.4.1 (Subdivision Tree) *Suppose f is an audio stream.*

1. ***Subdivision node.*** *Given an audio stream f, let us define a* subdivision node *to be a structure $\langle start, end, min, max, prev, next, child\rangle$ where*

   (a) $start$ and end are node's starting and ending quanta in f such that $start < end$,

   (b) $min(f)$ and $max(f)$ are lower and upper bounds on $\{f(start), \ldots, f(end-1)\}$,

   (c) $prev$ and $next$ are pointers to sibling nodes, such that $prev.end = start$ and $next.start = end$, and

   (d) $child$ is a pointer to the child node containing the starting quantum of the segment.

2. ***Subdivision.*** *A* subdivision *is a set of subdivision nodes* $c = \{N_1, \ldots, N_n\}$ *such that* $N_1.prev = NIL$, $N_i.prev = N_{i-1}$, $N_n.next = NIL$, *and* $N_i.next = N_{i+1}$. *A subdivision is characterized by its first and last nodes:* $first(c) = N_1$ *and* $last(c) = N_n$.

3. ***Subdivision tree.*** *Given a maximal utilization* ut_{max}, *a* subdivision tree *is a set of subdivisions* $T = \{c_1, \ldots, c_n\}$ *such that the root*

$$c_1 = \{\langle 0, length(f), min(f), max(f), NIL, NIL, first(c_1)\rangle\}$$

and for any subdivision node N *it is true that*

   (a) if $ut(N) \geq ut_{max}$ *then* $N.child = NIL$ *and*

   (b) otherwise, there is a subdivision $c_i \in T$ *such that* $N.child = first(c_i)$ *and* $\forall N' \in c_i\ ut(N') \geq 2 \cdot ut(N)$.

Example 3.4.2 (Building a Subdivision Tree) *Columns 5-6 of the Table 3.1 show steps taken by the* $SegmentStream()$ *algorithm called with* $ut_{min} = 2 \cdot 0.37 = 0.64$ *on each segment of the original subdivision obtained in the previous example. The table shows that each of the two original segments is further subdivided into segments* $\langle 0, 3, 6, 9\rangle$, $\langle 3, 15, 4, 10\rangle$, $\langle 15, 17, 11, 13\rangle$, $\langle 17, 19, 15, 16\rangle$, *and* $\langle 19, 38, 4, 17\rangle$. *These new segments make the second level of a subdivision tree whose first level consists of the original two segments from columns 3-4. Figure 3.6 shows the resulting subdivision tree.*

One can use the subdivision tree to search for stream values falling into a range $[v_{min}, v_{max}]$ with the following algorithm:

Algorithm FindInRange($Data$,N,q,v_{min},v_{max})
  $Data$ is the audio stream
  N is the subdivision node

```
    q is the starting quantum
    [v_min, v_max] is a range of values to search for
begin
    while N ≠ NIL and q ≥ N.end do N := N.next
    while N ≠ NIL do
        if [N.min, N.max] ∩ [v_min, v_max] ≠ ∅ then
            if N.child ≠ NIL then
                q' := FindInRange(N.child, q, v_min, v_max)
                if q' ≠ NotFound then return q'
            else
                for q' ∈ [max(q, N.start), N.end) do
                    if Data(q') ∈ [v_min, v_max] then return q'
                end for
            end if
        end if
        N := N.next
    end while
    return NotFound
end
```

Given a stream f, a starting quantum q, a range of interest $[v_{min}, v_{max}]$, and a subdivision tree T, the recursive $FindInRange()$ algorithm is called on the first node of T's root subdivision as

$$FindInRange(f, first(root(T)), q, v_{min}, v_{max}).$$

The algorithm starts by scanning the subdivision for the first node intersecting the quantum range $[q, +\infty)$, as we are only interested in quanta starting from q. It then looks for a node whose value range intersects $[v_{min}, v_{max}]$. If that node is subdivided further, $FindInRange()$ calls itself on this child subdivision. Otherwise, it resorts to direct scanning the audio stream within node's boundaries.

By replacing the $[N.min, N.max] \cap [v_{min}, v_{max}] \neq \emptyset$ statement with its negation, one can obtain the algorithm $FindOutRange()$ that searches for data *outside* of the range $[v_{min}, v_{max}]$.

Together, $FindInRange()$ and $FindOutRange()$ allow to find spans of quanta where stream values lie inside or outside a given range. One can use these algorithms

to accelerate the $SkipTo()$ API call. Then the $SkipNIL()$ API call can be implemented by looking for the data outside the $[dv(f), dv(f)]$ range.

3.4.1 Experiments

I have used a database of narrated Chaucer works, opera recordings (rendered from MIDI files), and other audio files totaling to about 80 million quanta, with individual recordings varying in length from several thousand to eight million quanta. Most recordings were two to four million quanta in length.

In the experiments, I assessed the effectiveness of the subdivision tree indexing. A *maximal amplitude* stream amp sampled over 10ms periods with the value range of $[0, 32767]$ has been used for experiments. The size of an index file, shown in Figure 3.7, has largely been under 5% of the data file size.

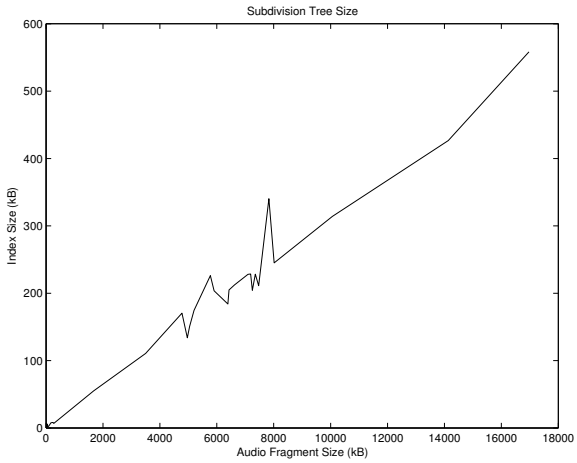


Figure 3.7: Subdivision Tree Size.

First, I ran SELECT queries

$$\sigma_{a.amp \geq v \wedge a.amp \leq 10000}(a),$$

where $v = 0, 2000, 4000, 6000, 8000$, five times for each audio file a in the database. Figure 3.8 shows average execution times for these queries with and without indexing, as a function of the data size. The first three graphs correspond to queries that select $[0, 10000]$ (highest selectivity), $[4000, 10000]$ (average selectivity), and $[8000, 10000]$ (lowest selectivity) amplitude ranges. As expected, index benefits grow as selectivity falls. The last graph shows the average execution time for all selectivities. It appears that the index improves SELECT performance by as much as seven times.

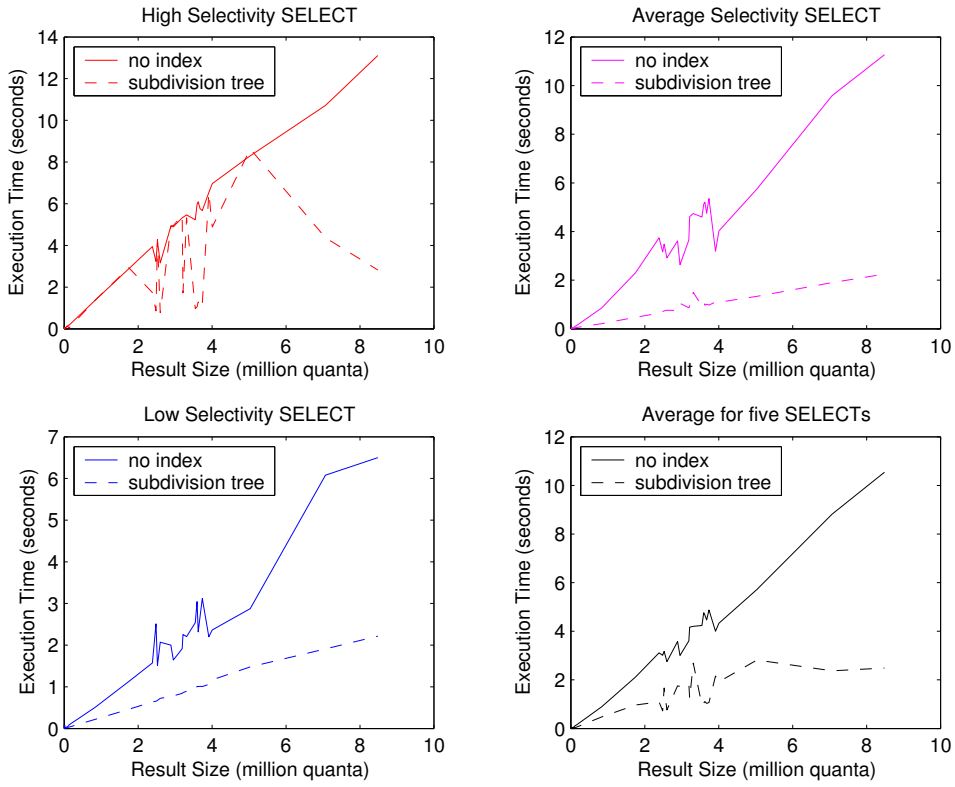


Figure 3.8: SubTree SELECT Performance.

Next, I ran MIX queries

$$a_1 \otimes_{a_1.amp \leq v \wedge a_2.amp > v} a_2,$$

where $v = 2000, 4000, 6000, 8000$, for all possible combinations of files a_1, a_2 in

the database. Figure 3.9 shows average execution times for these queries with and without indexing, as a function of the data size. The first three graphs correspond to queries with the "mixing threshold" of 2000, 4000, and 8000. The last graph shows the average execution time for all threshold values. While index benefits are less pronounced in this case, they are still clearly seen in all the graphs.

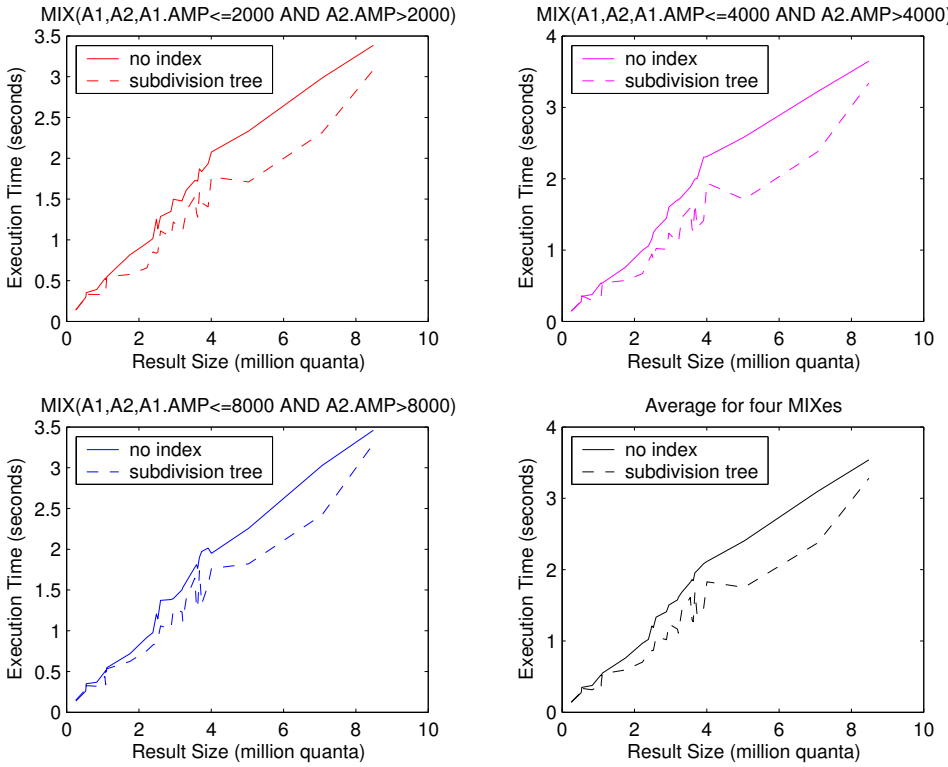


Figure 3.9: Mixing Performance.

3.5 Optimizing MATCH

Due to the commonality of pattern searches, the *matching operator* is probably the most important in the algebra. It is also most difficult to implement, given the complexity of the task and the huge amount of input data. Therefore, let us spend some

time looking at the implementation of this operator.

The problem of matching audio data has been tackled by many researchers, such as [33, 27, 69, 47, 65, 41]. Most of them look exclusively at *melodies* i.e. musical scores, represented with character strings, as opposed to waveforms or other audio representations. There are also works that match audio files based on feature vectors, returning the whole file if its vector matches, as opposed to finding a matching spot in that file [27].

I believe that to search audio effectively, one needs to use *all* available knowledge of an audio recording, such as its waveform, frequency spectrum, MIDI score, and text transcript, at once. The MATCH operator in ADA provides a formal foundation for such multi-stream matching. As *numeric* representations (such as waveform, amplitude, etc.) are most commonly available though, this section specifically concentrates on numeric stream matching. Approaches described here can be applied to a larger domain of audio data than melody-based approaches described above, but they are also complicated by the inherent irregularity of audio waveforms, their large size, and large "alphabets" of thousands of values. While lengths of musical scores rarely exceed a few thousand characters, waveforms easily reach millions of samples in length. Furthermore, distance metrics used in waveform matching (such as quadratic distance) require scanning the entire pattern to find its distance to the data fragment and thus greatly reduce benefits of suffix tries/trees (often used for musical score matching).

There is a lot of research in matching time series, such as stock quote history or sensor data [13, 12, 22, 66, 67, 87]. Such time series contain numeric data very similar to audio waveforms and are matched using quadratic, Manhattan, and maximal distance metrics, among others. Thus, approaches used to match time series can be

very useful when matching audio. Unfortunately, none of the above works specifically consider audio data. Also, time series used in the above works are usually limited to $10^2 - 10^6$ data points, much shorter than a typical audio file.

The ultimate purpose of matching is to find audio fragments that *sound similar* to the human ear. One way to do this would be to match waveform streams. Unfortunately, this approach is very sensitive to noise and phase difference between streams. To take care of these issues, one may compare *compound amplitude* streams instead, where sound amplitude is averaged or maximized over a period of n samples:

$$\begin{aligned} amp_{avg}(i) &= \frac{\Sigma_{j\in[-n/2,n/2)}|wave(i+j)|}{n}, \\ amp_{max}(i) &= max_{j\in[-n/2,n/2)}|wave(i+j)|. \end{aligned}$$

An example of the *maximum amplitude* stream amp_{max} is shown in Figure 3.10.

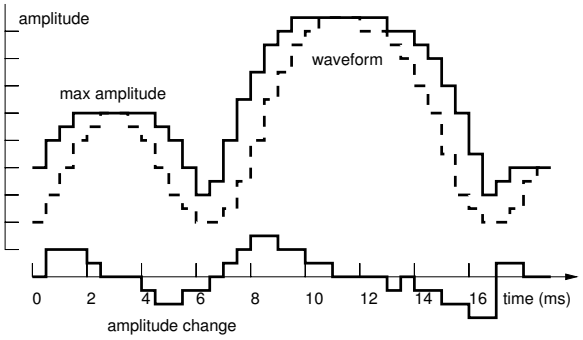


Figure 3.10: Maximum Amplitude and Amplitude Change Streams.

In addition, two streams may sound similar but have different volumes. Thus, we need some audio characteristics that do not change with the volume. One such characteristic is an amplitude change from one sample to the next relative to the total amplitude of these two samples:

$$ach(i) = \frac{amp(i+1) - amp(i)}{amp(i+1) + amp(i)}.$$

Assume $ach(i) = 0$ when $amp(i+1) + amp(i) = 0$. A sketch of the ach stream is shown in Figure 3.10.

The following algorithm uses the quadratic distance measure to find the best k pattern matches in an audio stream and returns their positions:

Algorithm NaiveFindPattern($Data$,$Pattern$,k,d_{max})
  $Data$ is an audio stream
  $Pattern$ is an audio stream such that $length(Pattern) \leq length(Data)$
  k is the number of best matches to return
  d_{max} is the distance threshold
  $List$ is a list of $\langle i, d \rangle$ pairs sorted by increasing d
begin
  $List := \{\langle 0, +\infty \rangle\}$
  for $i \in [0, length(Data) - length(Pattern)]$ do
    $d := 0$
    for $j \in [0, length(Pattern))$ do
      $d := d + (Data(i+j) - Pattern(j))^2$
      $d' := \frac{d}{length(Pattern) \cdot (max(Pattern) - min(Pattern))^2}$
      if $d' > d_{max}$ or ($size(List) = k$ and $d \geq tail(List).d$) then break
    end for
    if $d' \leq d_{max}$ and ($size(List) < k$ or $d < tail(List).d$) then
      $add(List, \langle i, d \rangle)$
      if $size(List) > k$ then $delete(List, tail(List))$
    end if
  end for
  return $\{[i, i + length(Pattern)) \mid \langle i, \_ \rangle \in List\}$
end

Notice that the $NaiveFindPattern()$ algorithm will terminate any pattern match as soon as its distance d exceeds the threshold d_{max} or the k^{th} best distance found so far. Aside from this optimization, the algorithm is fairly naive and has the time complexity of $O((length(Data) - length(Pattern)) \cdot length(Pattern))$.

It is clear that $NaiveFindPattern()$ is very time consuming, making naive matching unfeasible for large audio databases or even large standalone audio files. To accelerate matching, one can consider following approaches:

1. The $NaiveFindPattern()$ algorithm can be modified to operate directly on RLE compressed stream representations. If c_D and c_P are compression factors

for the data and pattern streams respectively, the algorithm complexity will be reduced by a factor of $O(c_D \cdot c_P)$.

2. By its nature, audio data tends to be smooth. It changes slowly with time, making it very likely that the values at adjacent quanta do not differ much. This is especially true for *compound amplitude* streams described in this section. Therefore, it may be possible to "downsample" streams by picking their values at every n^{th} quantum with the RESAMPLE operator, as shown in Figure 3.11, and match these downsampled streams instead of matching full streams. This reduces the matching complexity by a factor of $O(n^2)$, at the cost of reduced precision and recall, as will be shown in later experiments.

3. One can use a data structure, such as a *subdivision* or a variation of a *suffix trie* to index data and/or pattern and make use of this index to accelerate matching. This is the approach we are going to consider in a greater detail below.

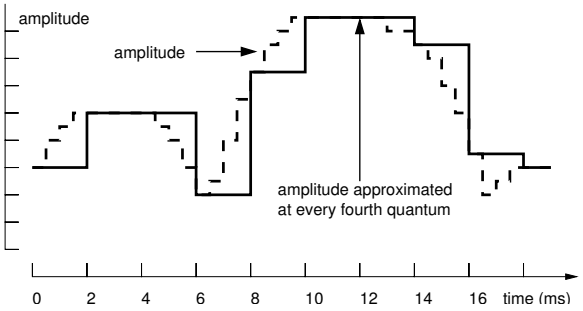


Figure 3.11: Downsampled Amplitude Stream.

3.5.1 Matching with Subdivisions

Let us start by representing both the audio and the pattern streams with subdivisions. A subdivision is similar to an RLE encoding but it uses rectangles to represent spans

of audio data where RLE is limited to line segments. Our problem then becomes to match rectangles taken from two subdivisions against each other. When using the quadratic distance to measure similarity between segments, one can make the following observation:

Proposition 3.5.1 (Bounds on Segment Distance) *Suppose we compute the quadratic distance between two segments s_1, s_2 of equal length ℓ as follows:*

$$dist(s_1, s_2) \quad = \quad \frac{1}{\ell} \cdot \sum_{i \in [0,l)} (s_1(i) - s_2(i))^2.$$

Then the upper and lower bounds on dist() are

$$dist(s_1, s_2) \quad \geq \quad (max(s_2.min - s_1.max, 0) + max(s_1.min - s_2.max, 0))^2,$$

$$dist(s_1, s_2) \quad \leq \quad max(s_2.max - s_1.min, s_1.max - s_2.min)^2.$$

A similar proposition holds for linear distance.

The proposition above is used in the following $MatchSegments()$ algorithm which, given data and pattern subdivisions and a distance limit d_{max}, returns a set of quantum ranges where distance between the data and the pattern is less than or equal to this limit:

Algorithm MatchSegments($Data$,$Pattern$,N_D,N_P,q_{min},q_{max},d_{max})
$Data$ is the data stream
$Pattern$ is the pattern stream
N_D is the first node of a data subdivision
N_P is the first node of a pattern subdivision
$[q_{min}, q_{max})$ is the time range to search
d_{max} is the maximal allowed distance
begin
$Result := \emptyset$
$N_{check} := NIL$
$N_{take} := NIL$
$t := max(q_{min}, N_D.start)$
// Find the first data node that intersects $[q_{min}, q_{max})$
while $N_D \neq NIL$ and $t \geq N_D.end$ do $N_D := N_D.next$
// Shift pattern across data until we reach the q_{max}
while $N_D \neq NIL$ and $t < q_{max}$ do

$P_1 := N_D$
$P_2 := N_P$
$\delta t := N_D.end - t$
$MIN := 0$
$MAX := 0$
// Traverse pattern subdivision comparing it against data
while $P_1 \neq NIL$ and $P_2 \neq NIL$ do
  // Compute MIN/MAX for the overlap of P_1/P_2
  $\ell := min(P_1.end, P_2.end + t) - max(P_1.start, P_2.start + t)$
  $MIN := MIN + \ell \cdot (max(P_2.min - P_1.max, 0) + max(P_1.min - P_2.max, 0))^2$
  $MAX := MAX + \ell \cdot (max(P_1.max - P_1.min, P_1.max - P_2.min))^2$
  // Shift to the next combination of P_1/P_2
  if $P_2.end + t > P_1.end$ then $P_1 := P_1.next$
  else
    if $P_2.end + t = P_1.end$ then $P_1 := P_1.next$ else $\delta t := min(\delta t, P_1.end - P_2.end - t)$
    $P_2 := P_2.next$
  end if
end while
// If pattern subdivision has been fully traversed...
if $P_2 = NIL$ then
  $MIN := \frac{MIN}{length(Pattern) \cdot (max(Pattern) - min(Pattern))^2}$
  $MAX := \frac{MAX}{length(Pattern) \cdot (max(Pattern) - min(Pattern))^2}$
  // If found start of a definite match...
  if $N_{take} = NIL$ and $MAX \leq d_{max}$ then
    $t_{take} := t_{prev} + (t - t_{prev}) \cdot \frac{MAX_{prev} - d_{max}}{MAX_{prev} - MAX}$
    if $t_{take} \geq N_D.start$ then $N_{take} := N_D$ else $N_{take} := N_D.prev$
    if $MIN_{prev} > d_{max}$ then
      $t'' := t_{prev} + (t - t_{prev}) \cdot \frac{MIN_{prev} - d_{max}}{MIN_{prev} - MIN}$
      $Result := Result \cup FindPattern(Data, Pattern, t'', t_{take}, d_{max})$
    end if
  end if
  // If found end of a definite match...
  if $N_{take} \neq NIL$ and $MAX > d_{max}$ then
    $t' := t_{prev} + (t - t_{prev}) \cdot \frac{MAX_{prev} - d_{max}}{MAX_{prev} - MAX}$
    $Result := Result \cup [t_{take}, t')$
    $N_{take} := NIL$
    if $MIN_{prev} > d_{max}$ then
      $t'' := t_{prev} + (t - t_{prev}) \cdot \frac{MIN_{prev} - d_{max}}{MIN_{prev} - MIN}$
      $Result := Result \cup FindPattern(Data, Pattern, t', t'', d_{max})$
    end if
  end if
  // If found start of a possible match...
  if $N_{check} = NIL$ and $d_{max} \in [MIN, MAX)$ then
    if $MIN_{prev} > d_{max}$ then
      $\ell := \frac{MIN_{prev} - d_{max}}{MIN_{prev} - MIN}$
    else
      $\ell := \frac{MAX_{prev} - d_{max}}{MAX_{prev} - MAX}$
    end if

```
            t_check := t_prev + ℓ · (t − t_prev)
            if t_check ≥ N_D.start then N_check := N_D else N_check := N_D.prev
        end if
        // If found end of a possible match...
        if N_check ≠ NIL and d_max ∉ [MIN, MAX) then
            if MIN_prev > d_max then
                ℓ := (MIN_prev − d_max) / (MIN_prev − MIN)
            else
                ℓ := (MAX_prev − d_max) / (MAX_prev − MAX)
            end if
            Result := Result ∪ FindPattern(Data, Pattern, t_check, t_prev + ℓ·(t − t_prev), d_max)
            P_check := NIL
        end if
        MIN_prev := MIN
        MAX_prev := MAX
        t_prev := t
        t := t + δt
    end if
  end while
  if N_take ≠ NIL then
    Result := Result ∪ [t_take, min(length(Data), q_max + length(Pattern)))
  end if
  if N_check ≠ NIL then
    Result := Result ∪ FindPattern(Data, Pattern, t_check, q_max, d_max)
  end if
  return Result
end
```

Given data and pattern streams $Data$ and $Pattern$ with corresponding subdivisions C_D and C_P, the $MatchSegments()$ algorithm is invoked as

$$MatchSegments(Data, Pattern, first(C_D), first(C_P), 0, length(Data), d_{max})$$

and slides pattern segments along data segments while computing the lower (MIN) and upper (MAX) bounds on the distance between the data and the pattern. Ranges of quanta where $MAX \leq d_{max}$ are immediately picked as answers, while ranges where $MIN > d_{max}$ are ignored. For all other ranges (where $MIN \leq d_{max} < MAX$) $MatchSegments()$ calls the naive $FindPattern()$ algorithm that scans actual data in these ranges, while avoiding the complete scan of data.

As shown before, the $NaiveFindPattern()$ performs the pattern matching for each subset of the data that has the same length as the pattern. The $MatchSegments()$

algorithm, on the other hand, only performs matching at quanta where data and pattern segment boundaries coincide, thus reducing the search complexity. For further efficiency, the $MatchSegments()$ algorithm can also be modified to work on subdivision trees.

3.5.2 Pattern Tree

So far, we have looked at matching pattern stream against a stream in a single audio file and found it to be a time consuming operation. The task becomes even more complicated when matching is done in an audio database containing a large number of audio files. To facilitate such operation, one needs an index that tells which database files may contain given pattern and which one can be safely omitted from the search.

When comparing audio pattern matching to other problems currently faced by computer scientists, two similar tasks immediately catch one's attention: the *text substring search* and the *genetic sequence search*. A well known method to cope with these problems is by using so-called *suffix tries and trees* [83] to index all substrings occurring in the data.

Unfortunately, audio pattern matching is somewhat different from both text and genetic searching. First of all, unlike text (with an alphabet of several dozen different characters) and genetic code (with an alphabet of just four characters), a typical audio stream has an "alphabet" of hundreds (8bit audio) or even thousands (16bit audio) of values. Thus, plain suffix trees tend to branch explosively when storing audio data.

Secondly, there is a difference in the distance measures. When matching substrings in a text or a genetic sequence, we are generally looking for the longest sub-

string that exactly matches the beginning of the pattern, i.e.

$$dist(t_1, t_2) = \frac{1}{card(\{t_1(i) \mid \forall 0 \leq j \leq i \, t_1(i) = t_2(i)\})}.$$

Alternatively, one may be looking for the longest common substring or use the least editing distance. All these measures are based on equivalence between characters, allowing matching algorithms to follow a tree path corresponding to these characters. Once characters from two sequences do not match, the path traversal terminates and the distance becomes known.

Unlike distance measures used in the text matching, waveform distance measures require computing the *difference* between corresponding characters taken from two sequences. The distance becomes known only after the entire pattern is traversed. This feature leads to multiple path traversals when looking for the closest match. Nevertheless, one can compute upper and lower bounds on the distance value while traversing each path and cut traversal short as soon as it becomes clear that the distance will be either smaller or larger than the given threshold.

Based on these specific properties of the audio stream matching, let us devise a structure inspired by the suffix trie that I will call the *pattern tree*.

Consider a numeric stream f such that $dom(f) = [-m, +m)$. Let us divide this range into b *buckets* $[l_1, u_1), \ldots, [l_b, u_b)$. Here, we can choose equal-sized buckets $[\frac{2mi}{b} - m, \frac{2m(i+1)}{b} - m)$, or notice that the audio data generally follows a normal distribution and use exponentially-sized buckets.

Consider now an ℓ-length pattern in f. Each quantum of this pattern belongs to a certain bucket, and the whole pattern can be indexed with a sequence of ℓ bucket numbers. One can compute the minimal and maximal distances from a given pattern f to all patterns indexed by this sequence:

$$MIN = \frac{\Sigma_{i=0}^{\ell-1} min((l_i - f(i))^2, (u_i - f(i))^2)}{\ell \cdot (max(f) - min(f))^2},$$

$$MAX \quad = \quad \frac{\Sigma_{i=0}^{\ell-1} max((l_i - f(i))^2, (u_i - f(i))^2)}{\ell \cdot (max(f) - min(f))^2}.$$

These bounds can be used to quickly find patterns that satisfy our matching requirements. Here is the *pattern tree* data structure:

Definition 3.5.1 (Pattern Tree) *Assume that we match single numeric streams whose values lie in the* $[-m, +m)$ *range. Given two positive integers* d, b *known as the tree* depth *and the* branching factor *respectively, the pattern tree consists of two types of nodes:*

1. *A* non-leaf node *is an array of node pointers* $\langle p_1, \ldots, p_b \rangle$ *where* p_i *corresponds to data values in the* $[\frac{2mi}{b} - m, \frac{2m(i+1)}{b} - m)$ *range.*

2. *A* leaf node *is a set of pointers to audio files that contain one or more occurrences of a pattern.*

Effectively, the pattern tree indexes d-length patterns occurring in audio files by providing b buckets for the value at each quanta. A non-leaf node at depth j in the pattern tree corresponds to the jth sample in a pattern. Depending on the value of this sample, the next node is chosen among the ones pointed to by the current node. At the bottom of a tree, there are leaf nodes with the lists of audio files that contain matched patterns.

One can notice that the maximal number of nodes in a pattern tree can reach $\frac{b^{d+2}-1}{b-1}$ and grows exponentially with the length of the indexed pattern d. The size of the tree can be reduced by matching slowly changing amp and ach streams and using the RESAMPLE operator to lower their sampling frequency from the original 22-44kHz to 10-500Hz.

Given a path $[l_1, u_1), \ldots, [l_n, u_n)$ in the pattern tree, one can compute the minimal and maximal distance between f and all patterns rooted at the tree node $[l_n, u_n)$ as

follows:

$$\begin{aligned} MIN &= \frac{\Sigma_{i=1}^{n} min((l_i - f(i-1))^2, (u_i - f(i-1))^2)}{length(f) \cdot (max(f) - min(f))^2}, \\ MAX &= MIN + 1 - \frac{n}{length(f)}. \end{aligned}$$

Given a threshold d_{max} on the distance between f and the data, one starts traversing a pattern tree. Every time the traversal encounters a node where $MAX \leq d_{max}$, all audio files rooted at this node are taken for examination with $NaiveFindPattern()$ or $MatchSegments()$ algorithms. Every time the traversal encounters a node where MIN exceeds d_{max}, this node and all its descendants are ignored. This search strategy is represented with the following algorithm:

Algorithm MatchPTree(N,f,d_{max},q,d_{min})
N is the node of a pattern tree
f is the pattern stream
d_{max} is the maximal allowed distance
q is the current quantum
d_{min} is the current lower bound on distance
begin
$Result := \emptyset$
while $i \in [0, branch(N))$ do
if $N.child_i \neq NIL$ then
$d := d_{min} + min((f(q) - \frac{2mi}{branch(T)} + m)^2, \frac{2m(i+1)}{branch(T)} - m - f(q) - 1)^2)$
$d' := \frac{d}{length(f) \cdot (max(f) - min(f))^2}$
if $d' \leq d_{max}$ then
if $d' + 1 - \frac{q}{length(f)} \leq d_{max}$ then
$Result := Result \cup \{f \mid f$ is an audio file rooted at $N.child_i\}$
else
if $q < length(f) - 1$ then
$Result := Result \cup MatchPTree(N.child_i, f, d_{max}, q + 1, d)$
end if
end if
end if
end if
end while
return $Result$
end

Given a pattern tree T and a pattern audio stream f, the recursive $MatchPTree()$

algorithm is invoked as

$$MatchPTree(root(T), f, d_{max}, 0, 0)$$

and returns a set of audio files that contain patterns at the d_{max} distance from f. The $MatchPTree()$ algorithm can also be modified to find nearest neighbors by using d_{max} to keep track of the smallest distance and discarding the last $Result$ every time d_{max} decreases.

3.5.3 Audio Range Tree

As the pattern tree classifies all patterns occuring in audio files, it can grow very large. An alternative data structure, called the *audio range tree* (AR-tree), can be used for pattern matching in audio files. Instead of storing entire patterns, the audio range tree stores *ranges of values* occuring in each pattern in a tree. Each leaf of this tree represents a certain value range and contains audio file descriptors and ranges of quanta where a pattern with this range occurs.

Definition 3.5.2 (Audio Range Tree)

- *Given a node size n, an* non-leaf node *is a collection of n tuples $\langle v_i^{min}, v_i^{max}, p_i, leaf_i \rangle$ where p_i is a pointer to the child node, $leaf_i = true$ if p_i points to a leaf node, or $false$ if p_i points to a non-leaf node, and $[v_i^{min}, v_i^{max}]$ is a range of pattern values represented by all nodes rooted at N.*

- *A* leaf node *is a set of tuples $\langle a, start, end \rangle$ where a is an audio file descriptor, and $[start, end)$ is a range of quanta in a.*

- *An* audio range tree *T is a set of nodes such that for any node $N \in T$, if $1 \leq i, j \leq n$ and $N.p_i$ is a non-leaf node then $[N.p_i.v_j^{min}, N.p_i.v_j^{max}] \subseteq$*

$[N.v_i^{min}, N.v_i^{max}]$. *For all nodes* $N \neq root(T)$*, it is also true that there exists* $N' \in T$ *such that* $N' \neq N$ *and there is* $N'.p_i = N$.

Given a maximal allowed distance d_{max}, one can search the AR-tree for audio files and ranges of quanta that *may* contain the given pattern. A sequential search algorithm can then be applied to these ranges, eliminating the need to search the entire database.

```
Algorithm ARFind(N,f,d_max)
    N is the audio range tree node
    f is the pattern stream
    d_max is the maximal allowed distance
begin
    Result := ∅
    for all ⟨v_i^min, v_i^max, leaf_i, p_i⟩ ∈ N such that p_i ≠ NIL do
        MIN := 0
        MAX := 0
        for j ∈ [0, length(f) − 1] do
            MIN := MIN + max(0, max(v_i^min − f(j), f(j) − v_i^max))^2
            MAX := MAX + max(f(j) − v_i^min, v_i^max − f(j))^2
            MIN' := MIN / (length(f)·(max(f)−min(f))^2)
            MAX' := MAX / (length(f)·(max(f)−min(f))^2)
            if MIN' > d_max then break
        end for
        if leaf_i = true then
            if MIN' ≤ d_max then Result := Result ∪ p_i
        else
            if MAX' ≤ d_max then
                Result := Result ∪ { all ranges rooted at p_i}
            else
                if MIN' ≤ d_max then
                    Result := Result ∪ ARFind(p_i, f, d_max)
                end if
            end if
        end if
    end for
    return Result
end
```

Given an AR-tree T and a pattern stream f, the recursive $ARFind()$ algorithm, initially invoked as

$$ARFind(root(T), f, d_{max}),$$

returns a set of $\langle a, start, end\rangle$ tuples corresponding to the suspected pattern match ranges. A sequential matching algorithm should then be applied to the data in these ranges.

The addition of data to the AR-tree faces problems similar to the same operation in R-trees. While I propose the following heuristic algorithm to add patterns, it leaves space for improvement:

Algorithm ARAdd(N,$Data$,v_{min},v_{max},$start$,end)
N is the audio range tree node
$Data$ is the audio stream
$[v_{min}, v_{max}]$ is the range of values being added
$[start, end)$ is the range of quanta being added
begin
if N is a leaf then
$N := N \cup \{\langle Data, start, end\rangle\}$
return
end if
Look for a non-leaf $\langle v_i^{min}, v_i^{max}, leaf_i, p_i\rangle \in N$ such that ...
... $[v_{min}, v_{max}] \subseteq [v_i^{min}, v_i^{max}]$ and $\frac{v_{max}-v_{min}}{v_i^{max}-v_i^{min}}$ is maximal
if $\langle v_i^{min}, v_i^{max}, leaf_i, p_i\rangle$ found then
$ARAdd(p_i, Data, v_{min}, v_{max}, start, end)$
return
end if
Look for a non-leaf $\langle v_i^{min}, v_i^{max}, leaf_i, p_i\rangle \in N$ such that ...
... $\frac{max(v_{max}, v_i^{max}) - min(v_{min}, v_i^{min})}{v_i^{max}-v_i^{min}}$ is minimal
if $\langle v_i^{min}, v_i^{max}, leaf_i, p_i\rangle$ found then
$ARAdd(p_i, Data, v_{min}, v_{max}, start, end)$
return
end if
Look for a leaf $\langle v_i^{min}, v_i^{max}, leaf_i, p_i\rangle \in N$ such that $v_{min} = v_i^{min}$ and $v_{max} = v_i^{max}$
if $\langle v_i^{min}, v_i^{max}, leaf_i, p_i\rangle$ found then
$ARAdd(p_i, Data, v_{min}, v_{max}, start, end)$
return
end if
Look for a slot $\langle v_i^{min}, v_i^{max}, leaf_i, p_i\rangle \in N$ such that $p_i = NIL$
if $\langle v_i^{min}, v_i^{max}, leaf_i, p_i\rangle$ found then
$v_i^{min} := v_{min}$
$v_i^{max} := v_{max}$
$leaf_i := true$
$p_i := \{\langle Data, start, end\rangle\}$
return
end if
Create a new node N'
$ARAdd(N', Data, v_{min}, v_{max}, start, end)$
Choose $\{t_1, \ldots, t_{n/2}\} \subset N$ such that ...

... $[min(v_{min}, min_{1 \leq j \leq n/2} t_j.v^{min}), max(v_{max}, max_{1 \leq j \leq n/2} t_j.v^{max})]$ is minimal
Move $\{t_1, \ldots, t_{n/2}\}$ to N'
Choose slot $\langle v_i^{min}, v_i^{max}, leaf_i, p_i \rangle \in N$ such that $p_i = NIL$
$v_i^{min} := min(v_{min}, min_{1 \leq j \leq n/2} t_j.v^{min})$
$v_i^{max} := max(v_{max}, max_{1 \leq j \leq n/2} t_j.v^{max})$
$leaf_i := false$
$p_i := N'$
return
end

Given an AR-tree T, a pattern length ℓ, and an audio stream f, the recursive $ARAdd()$ algorithm is invoked as

$$ARAdd(root(T), f, min_{1 \leq j < \ell} f(i+j), max_{1 \leq j < \ell} f(i+j), i, i+1)$$

for every $i \in [0, length(f))$.

3.5.4 Experiments

The first batch of experiments investigates the performance of approximate matching. The experiments were conducted on a set of audio files normalized to a uniform 8kHz frequency. I first matched amp streams of the three shortest files to every file in the set with the

$$\mu^k_{amp}(a, a', d_{max})$$

query, thus producing $3 \cdot 13 = 39$ results in each run. I then ran the "approximated query"

$$\gamma_{8kHz}(\mu^k_{amp}(\gamma_{f_{apx}}(a), \gamma_{f_{apx}}(a'), d_{max}))$$

(where $\gamma_{f_{apx}}(a)$ resamples audio file a to frequency f_{apx}) on the same data and computed its *precision* and *recall* w.r.t. the original query. The experiment has been performed for approximation frequencies $f_{apx} = 10Hz, 50Hz, 100Hz, 200Hz, 500Hz$, best item numbers $k = 1, 3, 5, 10$, and distance thresholds $d_{max} = 0.25, 0.5, 0.75$.

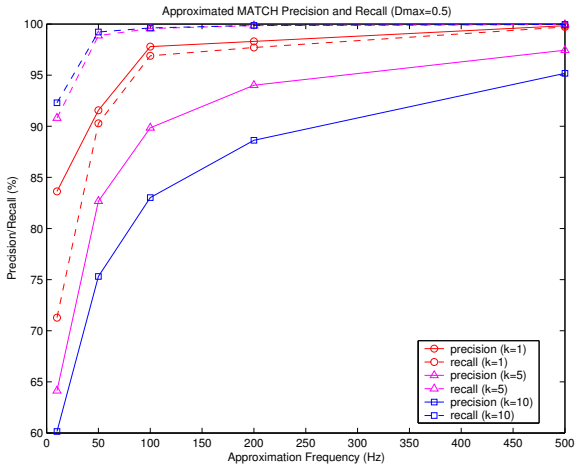


Figure 3.12: Precision and Recall as f_{apx} Changes.

The graph in Figure 3.12 shows how precision and recall depend on the approximation frequency. The graph shows that both precision and recall increase sharply as f_{apx} reaches 100-200Hz and then level off. It is also worth noting that, as the number of best matches k rises, precision becomes worse while recall becomes better. This is clearly visible from Figure 3.13 that shows precision and recall as functions of k. It has also been found that the choice of a distance threshold d_{max} does not affect the precision or recall.

Figure 3.14 reflects the benefits of approximated matching. The time to execute each query is plotted as a function of the audio length, with approximated queries taking very small time when compared to full queries.

The final result in Figure 3.15 shows approximated queries running on large (millions of quanta) audio files, averaged for several different patterns of about 13000-24000 quanta each. As expected, queries take longer to execute as the approximation frequency f_{apx} rises, but the difference is relatively small and even in the worst case ($f_{apx} = 500Hz$) it takes less than 19 seconds to process 80 million quanta of data.

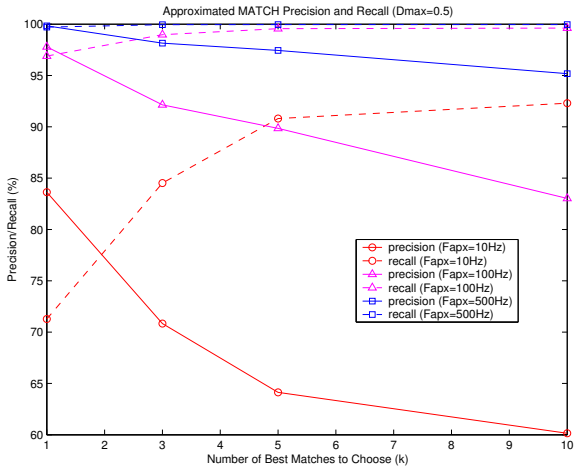


Figure 3.13: Precision and Recall as k Changes.

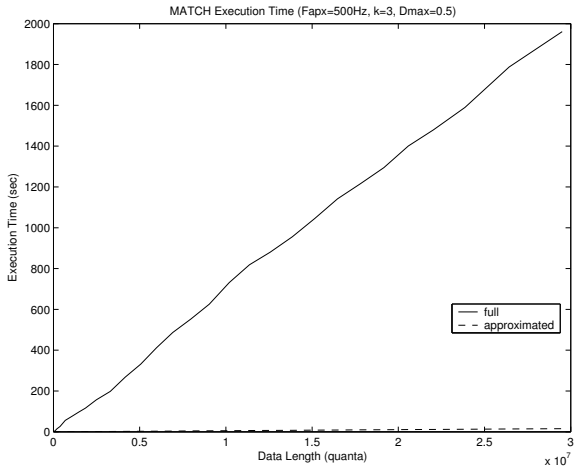


Figure 3.14: Matching Execution Times.

In the next experimental batch I have looked at how pattern trees and audio range trees improve matching performance. I started by approximating several million audio quanta in multiple audio files to the 100Hz frequency and indexing resulting patterns with a pattern tree. I then matched a query pattern against the tree and retrieved a list of audio files where the query pattern might occur. While the *full* query performed naive matching on *all* files in the database, the *accelerated* query only tried matching

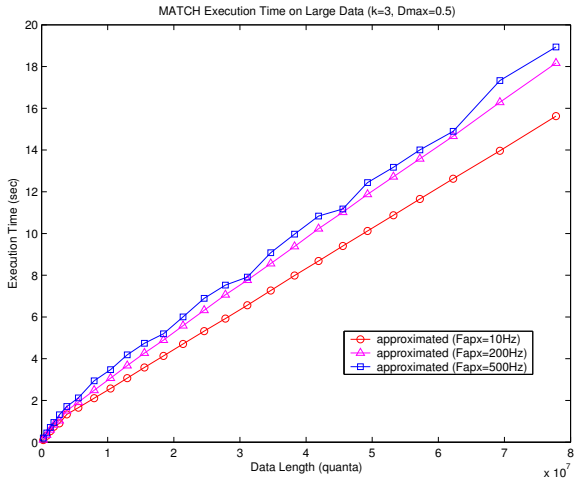


Figure 3.15: Approximated Matching on Large Data.

the files returned by the pattern tree search. Both queries ran on *non-approximated files*.

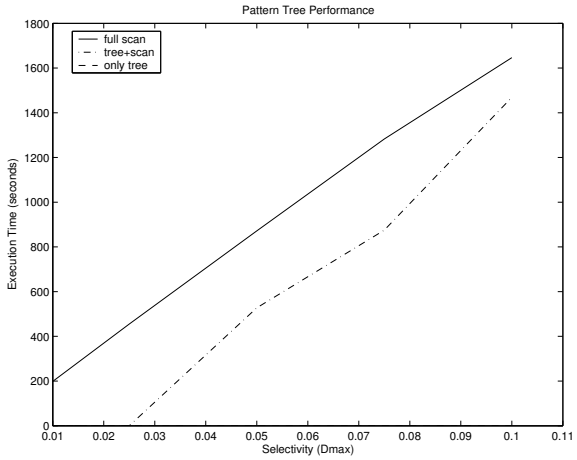


Figure 3.16: Pattern Tree Performance.

Figure 3.16 shows times taken by the full and accelerated queries as a function of the d_{max} value, with accelerated query taking 40% less time on the average. As expected, the benefits of the pattern tree decrease as the d_{max} grows and more patterns come under suspicion that they match the query pattern. Figure 3.16 also shows the

time taken by the tree search alone (without the following scan of data files), and it is negligibly small compared to the time taken by the scan.

My next step has been to implement the audio range tree and use it to index the same data as in the previous experiment, approximated to 500Hz. I then searched the tree for possible query pattern matches using different values of d_{max} and measured the percentage of data returned by this search relative to the full data size. The results, shown in Figure 3.17, indicate that the audio range tree allows to prune from 10% to 40% of the input data, with results improving as d_{max} and the data size grow.

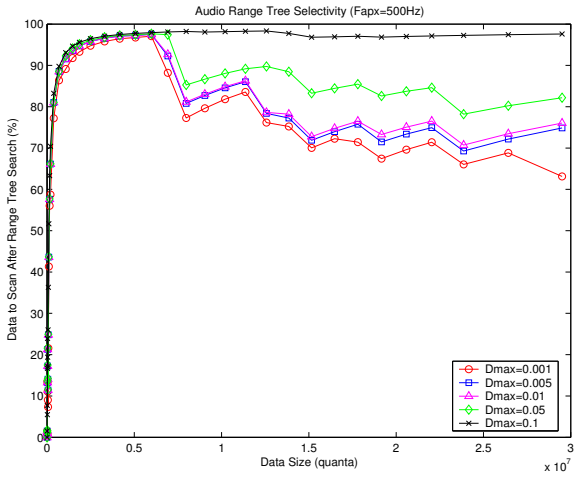


Figure 3.17: AR-Tree Pruning Performance.

In the final experiment of this batch, I compared the time required to perform the full scan of the database with the time required to search the audio range tree and then scan only the ranges of quanta returned by this search. The results of this experiment, shown in Figure 3.18, indicate that the audio range tree allows to save about 15-40% of the execution time, depending on the value of d_{max}.

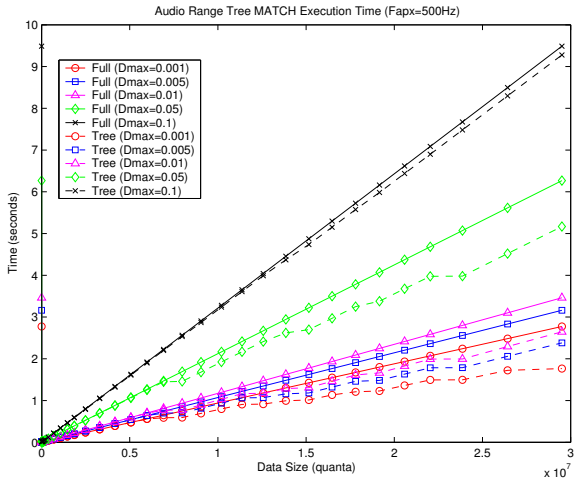


Figure 3.18: Audio Range Tree Performance.

3.6 System Implementation

The ADA Reference System is built as a collection of C++ classes corresponding to basic operators and data sources. Operator objects can be combined into query trees with data source objects at the bottom and executed. Due to potentially large amounts of intermediate data, the system tries to avoid storing intermediate results during query execution, although certain operators (such as MATCH) still require intermediate storage for efficiency.

Unlike the traditional relational algebra implementations that store all fields in a record at the same location, the ADA system is optimized to process audio streams stored in separate files. This allows us to avoid preprocessing the data (original .WAV or .AU files can be used), compress each stream in the best suitable way, and avoid accessing data not involved in queries.

The current ADA implementation is aware of the following audio streams:

- WAVE is a stream that contains the actual waveform, obtained from an audio file. All waveforms are converted to the `mono/16bit/signed/linear`

format.

- AMP is a maximum amplitude stream. For each quantum q, the value of this stream corresponds to the maximal absolute value of the WAVE stream in the $[q, q + \delta q)$ range of quanta, where δq usually corresponds to a period of $2 - 100ms$.

- FREQ is a set stream that carries the frequency spectrum. For each moment in time, this stream returns a mask of bits corresponding to different frequency ranges. Set bits correspond to spectrum components whose amplitudes are at least 70% of the maximum.

- KARAOKE is a text stream that contains syllables parsed from karaoke cue files. For each moment in time, this stream returns a syllable being sung at this moment, or the empty string if nothing is being sung.

3.6.1 The Algebraic Engine

The main C++ classes constituting the ADA algebraic engine are as follows:

- The Stream class describes a single audio stream characterized by the value at a given quantum, the default value, the length, and the quantum size. Concrete stream classes, such as WAVStream, AMPStream, and FRQStream, are derived from the Stream class.

- The Audio class describes a single audio file that is a collection of streams with the same length and quantum size.

- The Query class is derived from the Audio class and represents a single node in a query tree. Each object of this class has a type (SELECT , PROJECT , etc.) and

links to one or two inputs. The Query class supports such methods as $Create()$ (to execute query and write resulting streams to physical files) and $Clone()$ (to copy the entire query tree). Because each Query object is an instance of Audio, it also gives access to individual stream values in the query result. Very often, a query does not even need to be completely executed to access these values. Concrete query classes, such as QSelect, QProject, and QApply, are derived from the Query class.

- The QData class is derived from the Query class and describes a data source for a query. The QData nodes are always *leaves* in a query tree that supply query with the audio data. The basic implementation of QData reads data from physical files in a file system. Other implementations may use relational databases, network connections, or even audio recording hardware as sources.

In the most primitive case, the execution of a query comes down to scanning query inputs, performing algebraic operations, and creating output audio file(s). In reality though, one would like to use indices and skip over chunks of input that are of no interest to the query. The generic mechanism for such skipping is provided by API functions $SkipNIL()$ and $SkipTo()$, as described in the Section 3.4 of this chapter. Both these functions are available in the Stream, Audio, and Query classes. Classes that implement queries (QSelect, etc.) make automatic use of the input skipping functions, while Stream-derived classes optimize skipping if there is an index available or fall back to scanning otherwise.

3.6.2 The GUI

In addition to the algebraic engine, I have also created a simple GUI to browse through audio data, compose, and execute queries. While the ADA core is portable and can be used under both UNIX and Windows operating systems, the ADA GUI runs under Microsoft Windows and is written using the VCL toolkit in Borland C++ Builder. The GUI enables users to browse through audio files (Figure 3.19), match audio patterns against collections of audio files with varying precision (Figure 3.20) and visually design queries (Figure 3.21).

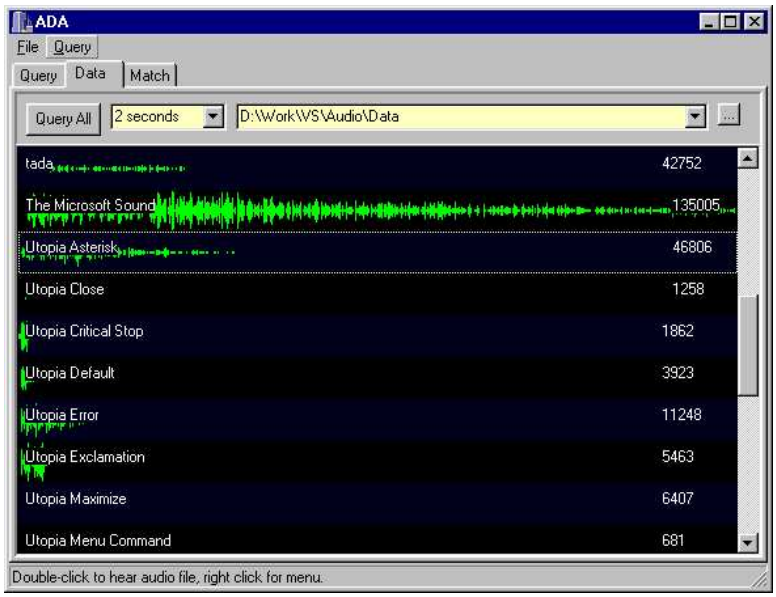


Figure 3.19: Data Browser GUI.

Figure 3.19 shows the audio data browser. The user selects a directory to browse and the length of audio to be displayed. The browser will then scan the directory for audio data and show found audio files in the window. The browser will also show graphically the requested length of a waveform from the beginning of each audio file. Users can left-click on audio files to play them. Right-clicking on an audio file brings

up a context menu allowing the user to use this file as a query input or a matching pattern.

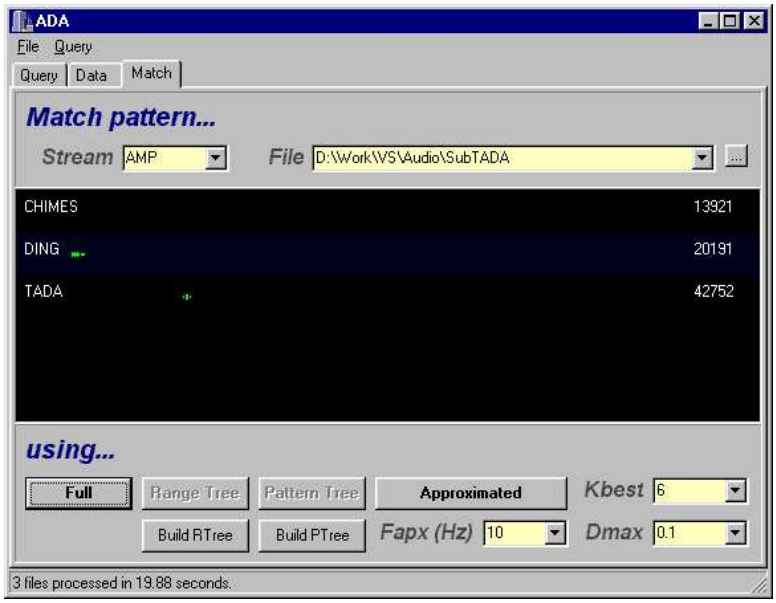


Figure 3.20: Pattern Matching GUI.

The matching function of the GUI, shown in the Figure 3.20, lets users select pattern audio files and search for matching patterns in the collection of audio files being currently browsed. The resulting files will be written to a separate directory and shown to the user. Users have control over the distance threshold d_{max} and the number k of best matches in each file. They also have an option to perform approximate matching by downsampling both pattern and data to the approximation frequency f_{apx}.

Finally, the query composition function, shown in the Figure 3.21, allows users to compose query trees by visual addition and deletion of operator nodes. Composed queries can be executed on either single audio files or their collections, or saved for future use. The WAVE stream that results from the execution of the *last* single audio file query is shown at the bottom and can be played with the usual walkman-like

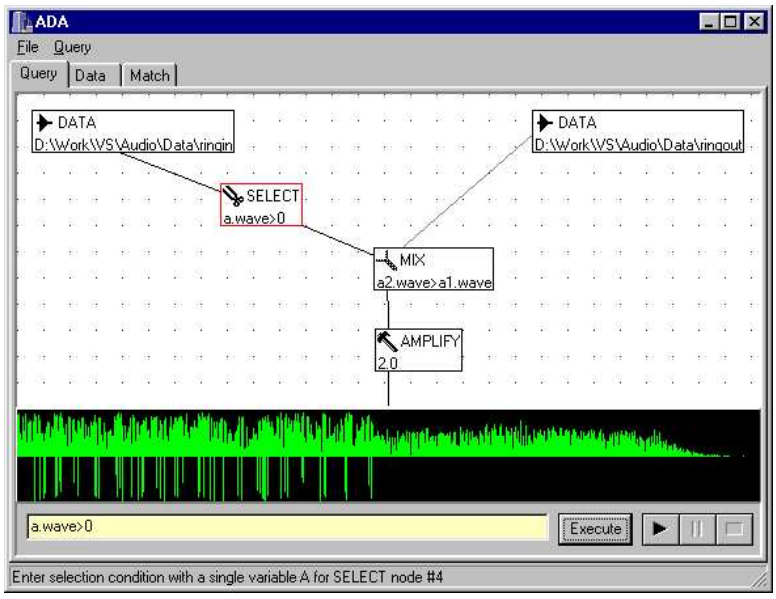


Figure 3.21: Query Composition GUI.

controls.

3.7 Related Work

There are relatively few works in audio databases. In research projects, audio is often used as a means for indexing the accompanying video, as exemplified by research conducted at Microsoft [39] using a database of lecture videos. The voice pitch of a lecturer was used to detect crucial presentation moments and segment the presentation by these moments. Using the pitch was only one factor though, others being information on slide changes and user feedback statistics.

As part of M.J. Witbrock and A.G. Hauptmann's *Informedia* project [85], an off-the-shelf speech recognition system called Sphinx-II was used to transcribe a collection of news broadcasts. The authors then evaluated the system performance for queries applied to (i) speech recognition results, (ii) phoneme recognition results, (iii)

closed captions converted to text, and (iv) manually prepared transcripts. Unfortunately, the results came out quite dismal for the speech recognition, both because of the large amount of recognition errors and the shortcomings of the text search mechanism.

Peter Schauble describes his work [71] on speech indexing and retrieval using so-called n-*grams*, where speech is converted to a sequence of phonemes. A sliding window is used to break the phoneme sequence into overlapping groups of n phonemes (where $n = 2 \ldots 4$) called n-grams. Audio documents are then indexed by a vector of the most characteristic n-grams. The system converts each query into n-grams, computes the corresponding vector and returns the closest matching documents from the database. This process helps avoiding speech recognition errors observed in [85] because a sequence of overlapping n-grams is much more tolerant of recognition errors than a sequence of single phonemes or the speech-recognized text.

Schauble further improves his search method by using n-gram representations of *words* instead of single n-grams for indexing. In both cases the index is built as an *inverted file* with keys corresponding to features (single n-grams or words) and whole audio documents are returned to a user. While this approach works for text documents which the user can evaluate at a glance, the linear nature of the audio makes such evaluation impossible. Thus, it seems necessary to cut the audio and only return the relevant parts to the user.

The ADA system attempts to provide a common platform upon which audio database projects like the ones described above can be built. The ADA is not limited to searching different audio representations though, but also contains operators to mix and otherwise transform these representations.

The problem of *matching* audio data has been tackled by many researchers, such

as [33, 27, 69, 47, 65, 41]. Most of exclusively deal with *melodies* i.e. musical scores as opposed to waveforms or other audio representations. For example, Ghias et al [33] convert MIDI scores to strings over a $\{U, D, S\}$ alphabet representing pitch changes ("up", "down", or "same"). The query is then literally *hummed* into a microphone, converted to a similar (albeit shorter) string using a variety of methods, and matched against the data string. Haus and Pollastri [65] expand the work done in [33] by refining the pitch-tracking method of converting hummed queries into MIDI scores.

The MUSIR system [69] uses both pitch and *duration* of notes by representing music with an alphabet made of MIDI note numbers and lengths. The query is input using a MIDI keyboard and matched to a database of melodies using n-grams. Lemström et al [47] also use a two-dimensional representation of musical data with *relative* pitch changes and note durations but match resulting strings using suffix-tries and the Boyer-Moore algorithm [10]. They show that using suffix-tries speeds up the queries, but the cost of constructing the trie exceeds its benefits. Finally, Hsu et al [41] provide an overall comparison of different approaches to music information retrieval based on the note duration and/or pitch and stored in lists or trees.

All works mentioned above attempt to search different representations of *music* as opposed to *audio* in general. This limits the domain of searchable documents to musical compositions, primarily stored in the MIDI format (although [33, 65] describe ways to recognize notes from an audio recording, for entering queries only). The only exception is a work by Foote [27] that extracts vectors of cepstral coefficients and uses them to classify audio files. The system implemented by Foote will only return whole audio files though. It is not able to pinpoint locations in these files where the query pattern occurs.

Unlike above works, ADA includes a generic way to match multiple audio rep-

resentations using different ways to measure similarity and pinpoint the exact spot where patterns occur in audio recordings. The choice of the indexing data structure will of course depend on the data type and the distance measure used.

The waveform matching described in this chapter is a special case of matching time series, such as stock quote history or sensor data, addressed in [13, 12, 22, 66, 67, 87]. The approach taken in these works requires computing an n-dimensional vector for each fixed-length subsequence of the data, using DFT coefficients [22, 67], DWT coefficients [13, 66], or piecewise mean values [12, 87]. The resulting vectors are then indexed using an R-tree or other data structure. Unfortunately, none of the above works specifically consider audio data. Also, time series used in the above works are usually limited to $10^2 - 10^6$ data points, much shorter than the length of a typical audio file.

3.8 Conclusions

Though there is a massive amount of audio data available in libraries, specialized archives, and on the web, there is little or no theory to query audio data. In this chapter, I have described a formal model of audio data and developed a relational model style algebra to query audio databases. I have proven a large number of equivalence results in this algebra that may be used for query optimization. I then looked at the data structures for indexing audio data and conducted experiments to show that proposed data structures significantly benefit query execution. Finally, I implemented a reference ADA system, complete with implementations of algebraic operators, command line interface, and the GUI.

Chapter 4

An Algebra For Video

4.1 Introduction

Video recordings can be found everywhere in the modern world, be it news, entertainment, sports, health care, family affairs, science, legal system, or security services. Fast accumulation of video data leads to the need to process, organize, search, and access this data. In the case of text and numeric data, these tasks have long been handled by *relational database management systems*. The goal of this chapter is to develop an algebra for operating on video data similar to the relational algebra.

The few existing video algebras either operate on *objects* occuring in frames as opposed to actual frames or segments of a video [59, 64, 17, 19] or use the hierarchical segment structure that complicates reasoning about algebraic operators [20]. In this chapter, I attempt to create a video algebra that supports both object and segment processing, yet it is sufficiently simple to reason about.

When talking about storing, searching, and processing motion video information, it is customary to think of it in terms of large continuous videos. In reality though, large parts of these videos may not contain any information useful to potential users.

A typical example of such situation is a surveilance video where nothing occurs on the screen for hours. Commercials in a TV broadcast may serve as another example of video segments that do not carry any useful data. Generally, each user may be interested in *his or her own* selection of video segments.

To avoid storing irrelevant content, one can use *video summaries* as opposed to the continuous video [24]. A summary is a set of video blocks (segments) characterized by their starting and ending times as well as their content. Blocks that carry irrelevant data can be dropped from the summary while the timing information is preserved in the remaining blocks. Any selection of blocks from a summary is also a summary. A continuous video is a special case of a summary that has no "gaps" from dropped blocks. Thus, while the entire video may be archived somewhere on a slow-access storage device (e.g. tape), the database system only needs to store a *subset* of this video relevant to the domain of system's operations. Using a database query language, users would select even smaller subsets reflecting their needs. Of course, to compose, store, search, and access such video summaries efficiently, it would be very helpful to have a common model and a *video database algebra*, much in the same way there is a relational algebra for operating on the relational data.

The algorithms for video summary creation will be thoroughfully discussed in Chapter 5. The present chapter describes the video database algebra (VDA) algebra that operates on videos and summaries made of blocks in a way similar to the relational algebra operating on tabular data. Section 4.2 defines the basic model for representing video data. Section 4.3 covers algebraic operators and equivalences. Section 4.4 discusses ways to accelerate operator execution by optimizing algorithms and using indices, followed by the experimental results. The cost model and the optimizer are discussed in Section 4.5. Section 4.6 describes the VDA system implementation.

Section 4.7 covers related works in video databases. Finally, Section 4.8 provides some concluding remarks.

4.2 Video Data Model

A video is a sequence of frames. Each frame can be characterized by a variety of properties, such as movement in some part of the frame, or the presence of a certain object or action. Let us call such properties *features* and define them formally:

Definition 4.2.1 (Feature) *A* feature *of a frame is a certain characteristic (such as color, text, or occurence of an object or action) pertaining to an area of a frame. Given a set τ, a feature of type τ is a structure $\langle name, val, loc, cov\rangle$ where $name$ is a string, val is a value from τ, loc is the region of a frame containing this feature, and $cov \in [0, 1]$ is the percentage of pixels in loc having this feature, also known as* coverage.

For any two features f_1, f_2, it is always true that if $f_1.name = f_2.name$ then both f_1 and f_2 have the same type τ, also known as $dom(name)$.

While loc may describe *any* connected frame region, for the sake of simplicity I will assume that $loc = \langle xl, yl, xh, yh\rangle$ is a *rectangle* with the right and bottom edges considered "open". Any other region can be approximated with its rectangular bounding box at the expense of decreasing its cov value.

To simplify the notation, I will often omit the cov member of the feature structure if $cov = 1$. The loc member may also be omitted. It is then assumed that loc corresponds to the entire frame.

For example, $\langle Human, true, \langle 10, 10, 50, 200\rangle, 0.7\rangle$ is a feature. So are $\langle Motion, true, \langle 60, 20, 90, 50\rangle\rangle$ and $\langle Caption, \text{“}NewYork\text{”}\rangle$. The last example carries no

location and coverage and therefore describes the entire frame with $cov = 1.0$. Other features may express annotations, color histograms, or the presence of certain objects, actions, or events.

Two features with the same name and value can be merged together in many ways two of which are shown in Figure 4.1 below. The first method, which I will call the *feature merge*, produces a single feature at the cost of precision:

Definition 4.2.2 (Feature Merge) *Given two features* f_1, f_2 *such that* $f_1.name = f_2.name$ *and* $f_1.val = f_2.val$*, the* merge *of these features is*

$$f_1 \oplus f_2 = \{\langle f_1.name, f_1.val, cov, bbox(f_1.loc \cup f_2.loc)\rangle\},$$

where

$$cov = \frac{f_1.cov \cdot area(f_1.loc) + f_2.cov \cdot area(f_2.loc) - area(f_1.loc \cap f_2.loc) \cdot (f_1.cov + f_2.cov)/2}{area(bbox(f_1.loc \cup f_2.loc))}.$$

The feature merge creates a bounding box around its arguments as shown at Figure 4.1, reconciles coverages by taking their average, and recomputes this resulting coverage with respect to the area of the bounding box. Thus, the feature merge leads to the loss of coverage precision.

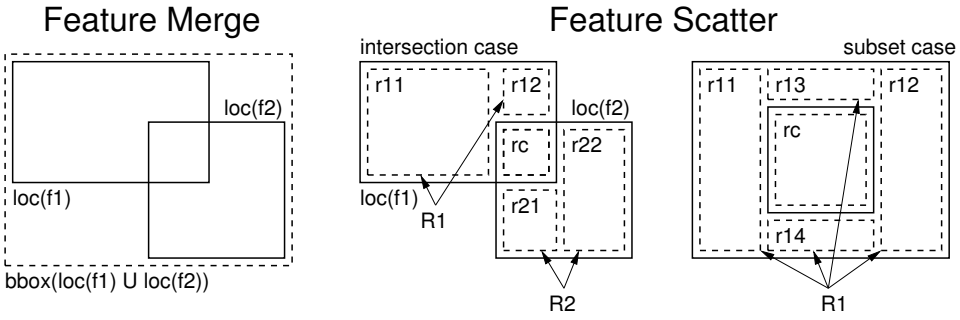


Figure 4.1: Feature Operations.

The second method, called the *feature scatter*, creates additional features while trying to retain coverage precision:

Definition 4.2.3 (Feature Scatter) *Given two features f_1, f_2 such that $f_1.name = f_2.name$, $f_1.val = f_2.val$, and $rc = f_1.loc \cap f_2.loc$, let R_1, R_2 be sets of disjoint rectangles such that $\bigcup_{r \in R_1} r = f_1.loc - rc$ and $\bigcup_{r \in R_2} r = f_2.loc - rc$. Then the* scatter *of f_1, f_2 w.r.t. R_1, R_2 is a set of features*

$$f_1 \otimes f_2 = \{\langle f_1.name, f_1.val, cov, loc\rangle \mid loc \in R_1 \cup R_2 \cup \{rc\}\}$$

where

$$cov = \begin{cases} \frac{f_1.cov + f_2.cov}{2} & if\ loc = rc \\ \frac{f_1.cov \cdot area(f_1.loc) - \frac{f_1.cov + f_2.cov}{2} \cdot area(rc)}{area(f_1.loc) - area(rc)} & if\ loc \in R_1 \\ \frac{f_2.cov \cdot area(f_2.loc) - \frac{f_1.cov + f_2.cov}{2} \cdot area(rc)}{area(f_2.loc) - area(rc)} & if\ loc \in R_2 \end{cases}$$

In the case of a feature scatter, we assume that the intersection rectangle rc has the average of two coverages $f_1.cov, f_2.cov$. The remaining pieces of $f_1.loc, f_2.loc$ are broken into rectangles as shown at Figure 4.1 and their coverages are recomputed with respect to the averaged coverage inside rc.

While R_1, R_2 may contain an arbitrary number of rectangles, it makes practical sense to minimize their cardinality, bringing it to at most *four* rectangles total in both R_1 and R_2, as shown by two cases at Figure 4.1. This also means that $card(f_1 \otimes f_2) \leq 5$.

Example 4.2.1 (Feature Merge/Scatter) *Consider two features:*

$$\begin{aligned} f_1 &= \langle Green, 70, \langle 50, 50, 100, 100\rangle, 0.7\rangle, \\ f_2 &= \langle Green, 70, \langle 70, 80, 120, 110\rangle, 0.5\rangle. \end{aligned}$$

The merge *of these two features will be a singleton set $f_1 \oplus f_2 = \{\langle Green,\ 70,\ \langle 50, 50, 120, 110\rangle,\ cov\rangle\}$ with the cov value computed as follows:*

$$\begin{aligned} area(f_1.loc) &= (100 - 50) \cdot (100 - 50) = 2500 \\ area(f_2.loc) &= (120 - 70) \cdot (110 - 80) = 1500 \end{aligned}$$

$$
\begin{aligned}
area(f_1.loc \cap f_2.loc) &= (100-70)\cdot(100-80) = 600 \\
area(bbox(f_1.loc \cup f_2.loc)) &= (120-50)\cdot(110-50) = 4200 \\
cov &= \frac{0.7\cdot 2500 + 0.5\cdot 1500 - 600\cdot(0.5+0.7)/2}{4200} \approx 0.51
\end{aligned}
$$

The scatter *of f_1 and f_2 will be a set of at least five features:*

$$
\begin{aligned}
f_1 \otimes f_2 = \{ & \\
& \langle Green, 70, \langle 50, 50, 70, 100\rangle, 0.7\rangle\}, \\
& \langle Green, 70, \langle 70, 50, 100, 80\rangle, 0.7\rangle\}, \\
& \langle Green, 70, \langle 100, 80, 120, 110\rangle, 0.5\rangle\}, \\
& \langle Green, 70, \langle 70, 100, 100, 110\rangle, 0.5\rangle\}, \\
& \langle Green, 70, \langle 70, 80, 100, 100\rangle, 0.6\rangle\} \\
& \}
\end{aligned}
$$

Sets of features can be merged with respect to a feature operation such as merge or scatter:

Definition 4.2.4 (Feature Set Merge) *Let's first define an "*int*" binary relation on features:*

$$int(f_1, f_2) \equiv f_1.name = f_2.name \wedge f_1.val = f_2.val \wedge f_1.loc \cap f_2.loc \neq \emptyset.$$

Given two sets of features F_1, F_2, the merge *of these sets with respect to a feature operation op (such as merge or scatter) is a new feature set*

$$
\begin{aligned}
F_1 \cup_{op} F_2 &= \{x \in F_1 \mid \forall x' \in F_2 : \neg int(x, x')\} \\
&\cup \{x \in F_2 \mid \forall x' \in F_1 : \neg int(x, x')\} \\
&\cup \{x \; op \; x' \mid x \in F_1 \wedge x' \in F_2 \wedge int(x, x')\}.
\end{aligned}
$$

A *block* is a piece of a video composed of several successful frames. It is characterized by the start and end times, the video data (i.e. the actual sequence of frame images) and a set of features found in its frames:

Definition 4.2.5 (Block) *A video* block *is a structure* $b = \langle t_s, t_e, FS, DS \rangle$ *where* $b.t_s$ *is the starting time,* $b.t_e > b.t_s$ *is the ending time,* $b.DS$ *is the video data (i.e. actual sequence of images), and* $b.FS$ *is a set of features occuring in* b*, with coverages computed with respect to the entire block (i.e. a number of frames).*

While it is not required by the block definition, I will assume that t_s and t_e represent frame numbers and therefore are integer. The *length* of a block $length(b) = b.t_e - b.t_s$. Two or more blocks can be merged together, producing a bigger block:

Definition 4.2.6 (Block Merge) *The* merge *of two blocks,* b_1 *and* b_2*, with respect to a feature operation* op *(such as merge or scatter), is a new block*

$$b_1 \cup_{op} b_2 = \langle min(b_1.t_s, b_2.t_s), max(b_1.t_e, b_2.t_e), b_1.FS \cup_{op} b_2.FS, DS \rangle,$$

where DS *computation is implementation-dependent.*

The block merge operator can be applied to more than two blocks. As blocks are effectively three-dimensional (horizontal, vertical, and time), merge and scatter operations must recompute coverages using all three dimensions when applied to block feature sets. The DS computation algorithm is left to the system implementor.

Now, when we have all the building blocks, let us define a *video*:

Definition 4.2.7 (Video) *A* video *is a finite sequence of blocks* $v = \{b_1, \ldots, b_n\}$ *such that* $\forall 1 \leq i < n : \; b_{i+1}.t_s = b_i.t_e$.

Given the above definition of a video, we can define a total ordering on blocks in a video as $\forall 1 \leq i, j \leq n : \ b_i \leq b_j \leftrightarrow b_i.t_s \leq b_j.t_s$. The *length* of a video $length(v) = \sum_{b_i \in v} length(b_i)$. A *video summary* is a subset of a video:

Definition 4.2.8 (Summary) *A* summary *of a video v is a sequence of blocks $s = \{b_1, \ldots, b_n\}$ such that $s \subseteq v$ and $\forall 1 \leq i < n : \ b_{i+1}.t_s \geq b_i.t_e$.*

Given the above definitions of videos and summaries, one can easily show that any video is also a summary of itself and that the total ordering defined for videos also applies for summaries. The *length* of a summary $length(s) = \sum_{b_i \in s} length(b_i)$.

We are finally ready to define a *video database*, which is simply a collection of summaries.

Definition 4.2.9 (Video Database) *A* video database VDB *is a set of video summaries.*

4.3 Algebraic Operators

Before moving on to definitions of algebraic operators, let us look at *selection conditions* and how they are evaluated. Selection conditions are used in many algebraic operators to choose certain blocks and features from the input.

4.3.1 Selection Conditions

Assume the existance of a set C_f of all possible features and a set V_f of all variables ranging over C_f. Similarly, there is a set C_b of all possible blocks and a set V_b of all variables ranging over C_b. For clarity, I will use the small b_* (f_*) to denote members of C_b (C_f) and the capital B_* (F_*) to denote members of V_b (V_f).

Definition 4.3.1 (Term) *(i) Any member of a set τ is a* term *of type τ. (ii) Any block constant or variable is a block term. (iii) Any feature constant or variable is a feature term.*

Definition 4.3.2 (Feature Atoms)

- *$TRUE$ and $FALSE$ are* feature atoms.
- *If f_1, f_2 are feature terms and "$\sim$" is a binary relation defined on feature values then $f_1 \sim f_2$ is a* feature atom.
- *If f is a feature term and str is string then $name(f, str)$ is a* feature atom.
- *If f_1, f_2 are feature terms then $inside(f_1, f_2)$, $overlap(f_1, f_2)$, $above(f_1, f_2)$, $below(f_1, f_2)$, $leftof(f_1, f_2)$, and $rightof(f_1, f_2)$ are all* feature atoms.

Definition 4.3.3 (Feature Conditions) *Every feature atom is a feature condition. If C_1, C_2 are feature conditions then so are $C_1 \wedge C_2$, $C_1 \vee C_2$, and $\neg C_1$. A* feature selection condition *(FSC) is a feature condition containing a variable $F \in V_f$. A* local FSC *is an FSC that contains no variables other than F. A* feature join condition *(FJC) is a feature condition containing exactly two variables $F_1, F_2 \in V_f$.*

Definition 4.3.4 (Feature Condition Interpretation) *Suppose f_1, f_2 are ground feature terms and C_1, C_2 are ground feature conditions. Let us define a function λ that takes a ground feature condition and returns a real value in the $[0, 1]$ range. The result of λ is computed in the following way:*

$$\lambda(f_1 \sim f_2) = \begin{cases} \frac{f_1.cov \cdot f_2.cov \cdot area(f_1.loc \cap f_2.loc)}{min(area(f_1.loc), area(f_2.loc))} & if\ f_1.name = f_2.name \wedge f_1.val \sim f_2.val \\ 0 & otherwise \end{cases}$$

$$\lambda(name(f_1, str)) = \begin{cases} 1 & if\ f_1.name = str \\ 0 & otherwise \end{cases}$$

$$\lambda(inside(f_1, f_2)) = \begin{cases} 1 & if\ f_1.loc \subseteq f_2.loc \\ 0 & otherwise \end{cases}$$

$$\lambda(overlap(f_1, f_2)) = \begin{cases} 1 & if\ f_1.loc \cap f_2.loc \neq \emptyset \\ 0 & otherwise \end{cases}$$

$$
\begin{aligned}
\lambda(above(f_1, f_2)) &= \begin{cases} 1 & if\ f_1.loc.yh \leq f_2.loc.yh \wedge f_1.loc.yl \leq f_2.loc.yl \\ 0 & otherwise \end{cases} \\
\lambda(below(f_1, f_2)) &= \begin{cases} 1 & if\ f_1.loc.yh \geq f_2.loc.yh \wedge f_1.loc.yl \geq f_2.loc.yl \\ 0 & otherwise \end{cases} \\
\lambda(leftof(f_1, f_2)) &= \begin{cases} 1 & if\ f_1.loc.xh \leq f_2.loc.xh \wedge f_1.loc.xl \leq f_2.loc.xl \\ 0 & otherwise \end{cases} \\
\lambda(rightof(f_1, f_2)) &= \begin{cases} 1 & if\ f_1.loc.xh \geq f_2.loc.xh \wedge f_1.loc.xl \geq f_2.loc.xl \\ 0 & otherwise \end{cases} \\
\lambda(C_1 \wedge C_2) &= min(\lambda(C_1), \lambda(C_2)) \\
\lambda(C_1 \vee C_2) &= max(\lambda(C_1), \lambda(C_2)) \\
\lambda(\neg C_1) &= 1 - \lambda(C_1) \\
\lambda(TRUE) &= 1 \\
\lambda(FALSE) &= 0
\end{aligned}
$$

Block atoms and conditions are similar to feature atoms and conditions but they operate on blocks.

Definition 4.3.5 (Block Atoms)

- $TRUE$ *and* $FALSE$ *are* block atoms.
- *If* b *is a block term and* C *is a feature condition then* $in(b, C)$ *is a* block atom.
- *If* b_1, b_2 *are block terms and* $n \in [0, +\infty)$ *is an integer number,* $within(b_1, b_2, n)$, $before(b_1, b_2, n)$, $after(b_1, b_2, n)$, $overlap(b_1, b_2)$, *and* $inside(b_1, b_2)$ *are* block atoms.

Definition 4.3.6 (Block Conditions) *Every block atom is a block condition. If* C_1, C_2 *are block conditions then so are* $C_1 \wedge C_2$, $C_1 \vee C_2$, *and* $\neg C_1$. *A* block selection condition *(BSC) is a block condition containing a variable* $B \in V_b$. *A* local BSC *is a BSC containing no variables other than* B. *A* block join condition *(BJC) is a block condition with exactly two variables* $B_1, B_2 \in V_b$.

Definition 4.3.7 (Block Condition Interpretation) *Suppose* b_1, b_2 *are ground block terms,* F *is a feature variable,* C *is a feature selection condition, and* C_1, C_2 *are*

ground block conditions. Let us define a function λ that takes a ground block condition and returns a real value in the $[0, 1]$ range. The result of λ is computed in the following way:

$$
\begin{aligned}
\lambda(in(b_1, C)) &= max_{all\ \theta}(\lambda(C\theta)) \\
\lambda(inside(b_1, b_2)) &= \begin{cases} 1 & if\ b_1.t_s \geq b_2.t_s \wedge b_1.t_e \leq b_2.t_e \\ 0 & otherwise \end{cases} \\
\lambda(overlap(b_1, b_2)) &= \begin{cases} 1 & if\ (b_1.t_s, b_1.t_e) \cap (b_2.t_s, b_2.t_e) \neq \emptyset \\ 0 & otherwise \end{cases} \\
\lambda(before(b_1, b_2, n)) &= \begin{cases} (n - b_1.t_s + b_2.t_e)/n & if\ 0 \leq b_1.t_s - b_2.t_e \leq n \\ 0 & otherwise \end{cases} \\
\lambda(after(b_1, b_2, n)) &= \lambda(before(b_2, b_1, n)) \\
\lambda(within(b_1, b_2, n)) &= max(\lambda(before(b_1, b_2, n)), \lambda(after(b_1, b_2, n))) \\
\lambda(C_1 \wedge C_2) &= min(\lambda(C_1), \lambda(C_2)) \\
\lambda(C_1 \vee C_2) &= max(\lambda(C_1), \lambda(C_2)) \\
\lambda(\neg C_1) &= 1 - \lambda(C_1) \\
\lambda(TRUE) &= 1 \\
\lambda(FALSE) &= 0
\end{aligned}
$$

Here, θ represents a variable substitution that grounds all variables in C by replacing them with distinct *features from $b_1.FS$ (i.e. no two variables are assigned to the same feature).*

4.3.2 The SELECT Operator

Similarly to the relational selection choosing rows from a table, the SELECT operator in VDA chooses from its input summary blocks that satisfy the selection condition:

Definition 4.3.8 (SELECT Operator) *Given a video summary v, a block selection condition C, and a real value $p_{min} \in [0,1]$, the* SELECT *operator $\sigma_C(v, p_{min})$ produces a new summary*

$$\sigma_C(v, p_{min}) = \{b \in v \mid max_{all\ \theta}(\lambda(C[B/b]\theta)) \geq p_{min}\},$$

where θ is a variable substitution that grounds all remaining variables in $C[B/b]$ by replacing them with distinct blocks from v.

The p_{min} argument to SELECT allows to control how strictly selected blocks satisfy the condition. In most practical applications, this parameter can be assumed to be a very small number ϵ. In such cases, I will omit it from the notation.

Here is a typical example of SELECT being used to find blocks where screen captions appear: $\sigma_{in(B,name(F,Caption))}(v)$. Here is another SELECT , detecting motion at two spots of the screen:

$$\sigma_{in(B,F=\langle Motion,true,\langle 50,50,100,75\rangle\rangle \vee F=\langle Motion,true,\langle 500,400,550,450\rangle\rangle)}(v).$$

Just like in the relational algebra, the order of SELECT operators can be changed without affecting the result, but only if their selection conditions are local:

Theorem 4.3.1 (Swapping SELECT Operators) *Given a video summary v, two local BSCs C_1, C_2 and two real values $p_1, p_2 \in [0,1]$, it is true that*

$$\sigma_{C_1}(\sigma_{C_2}(v, p_2), p_1) = \sigma_{C_2}(\sigma_{C_1}(v, p_1), p_2).$$

Proof of Theorem 4.3.1. Due to locality of C_1, C_2 and definition of SELECT , both SELECT operators consider their inputs one block at a time and either select this block or drop it, based on a single value of λ. Then let us consider an arbitrary block $b \in v$ and see if it is present in $v' = \sigma_{C_1}(\sigma_{C_2}(v, p_2), p_1)$ and $v'' = \sigma_{C_2}(\sigma_{C_1}(v, p_1), p_2)$, case by case:

1. $\lambda(C_1[B/b]) < p_1$ and $\lambda(C_2[B/b]) < p_2$ then $b \notin \sigma_{C_1}(v, p_1)$, $b \notin \sigma_{C_2}(v, p_2)$, and therefore $b \notin v'$ and $b \notin v''$.

2. If $\lambda(C_2[B/b]) \geq p_2$ and $\lambda(C_1[B/b]) < p_1$ then $b \in \sigma_{C_2}(v, p_2)$ but $b \notin v'$, as it is deleted by the outer SELECT operator. On the other hand, $b \notin \sigma_{C_1}(v, p_1)$ and therefore $b \notin v''$.

3. If $\lambda(C_1[B/b]) \geq p_1$ and $\lambda(C_2[B/b]) < p_2$ then $b \in \sigma_{C_1}(v, p_1)$ but $b \notin v''$, as it is deleted by the outer SELECT operator. On the other hand, $b \notin \sigma_{C_2}(v, p_2)$ and therefore $b \notin v'$.

4. If $\lambda(C_1[B/b]) \geq p_1$ and $\lambda(C_2[B/b]) \geq p_2$ then $b \in \sigma_{C_1}(v, p_1)$, $b \in \sigma_{C_2}(v, p_2)$, and also $b \in v'$ and $b \in v''$.

Thus, it has been shown that wherever $b \in v'$, it is also true that $b \in v''$ and vice versa. Therefore, $v' = v''$.

4.3.3 The PROJECT Operator

In the traditional relational algebra, the projection is used to select columns from a table. In VDA , features act as columns, while blocks play the role of rows. Thus, the PROJECT operator serves to select requested features in all input blocks:

Definition 4.3.9 (PROJECT Operator) *Given a video summary v, a feature selection condition C, and a real value $p_{min} \in [0, 1]$, the* PROJECT *operator $\pi_C(v, p_{min})$ produces a new summary*

$$\pi_C(v, p_{min}) = \{\langle b.t_s, b.t_e, FS, b.DS\rangle \mid \forall b \in v \; FS = \{f \in b.FS \mid max_{all\ \theta}(\lambda(C[F/f]\theta)) \geq p_{min}\}\},$$

where θ is a variable substitution that grounds all remaining variables in $C[F/f]$ by replacing them with distinct features from $b.FS$.

Same as with the SELECT operator, if p_{min} is omitted, it is assumed that $p_{min} = \epsilon$. Under certain conditions, two PROJECT operators can be swapped:

Theorem 4.3.2 (Swapping PROJECT Operators) *Given a video summary v, two local FSCs C_1, C_2 and two real values $p_1, p_2 \in [0, 1]$, it is true that*

$$\pi_{C_1}(\pi_{C_2}(v, p_2), p_1) = \pi_{C_2}(\pi_{C_1}(v, p_1), p_2).$$

Proof of Theorem 4.3.2. Let us consider two queries $v' = \pi_{C_1}(\pi_{C_2}(v, p_2), p_1)$ and $v'' = \pi_{C_2}(\pi_{C_1}(v, p_1), p_2)$. Due to definition of PROJECT , both PROJECT operators consider their inputs one block at a time and do not delete or add any blocks, i.e. for every block $b \in v$, there are blocks $b' \in v'$ and $b'' \in v''$. Furthermore, due to locality of C_1, C_2, each feature set $b.FS$ isz also considered one feature at a time. Then let us look at an arbitrary feature $f \in b.FS$ and see if it is present in $b'.FS$ and $b''.FS$, case by case:

1. $\lambda(C_1[F/f]) < p_1$ and $\lambda(C_2[F/f]) < p_2$ then f is deleted by inner PROJECT operators in both v', v'', and therefore $f \notin b'.FS$ and $f \notin b''.FS$.

2. If $\lambda(C_2[F/f]) \geq p_2$ and $\lambda(C_1[F/f]) < p_1$ then f is deleted by the outer PROJECT operator in v' and by the inner PROJECT operator in v''. Thus, $f \notin b'.FS$ and $f \notin b''.FS$.

3. If $\lambda(C_1[F/f]) \geq p_1$ and $\lambda(C_2[F/f]) < p_2$ then f is deleted by the inner PROJECT operator in v' and by the outer PROJECT operator in v''. Thus, $f \notin b'.FS$ and $f \notin b''.FS$.

4. If $\lambda(C_1[F/f]) \geq p_1$ and $\lambda(C_2[F/f]) \geq p_2$ then f is not deleted at all, and therefore $f \in b'.FS$ and $f \in b''.FS$.

Thus, we have shown that wherever $f \in b'.FS$, it is also true that $f \in b''.FS$ and vice versa. As it is true for any block $b \in v$ and any feature $f \in b.FS$, we can say that $v' = v''$.

PROJECT and SELECT can also be swapped, given that SELECT is not affected by changing feature content:

Theorem 4.3.3 (Swapping PROJECT and SELECT) *Given a video summary v, two real values $p_1, p_2 \in [0, 1]$, an FSC C_1, and a BSC C_2 that contains no $in()$ atoms, it is true that*

$$\pi_{C_1}(\sigma_{C_2}(v, p_2), p_1) = \sigma_{C_2}(\pi_{C_1}(v, p_1), p_2).$$

Proof of Theorem 4.3.3. Let us consider two queries $v' = \pi_{C_1}(\sigma_{C_2}(v, p_2), p_1)$ and $v'' = \sigma_{C_2}(\pi_{C_1}(v, p_1), p_2)$. Our task is to show that for every $b \in v$, if there is a corresponding block $b' \in v'$, there is also $b'' \in v''$ such that $b' = b''$, and vice versa. Due to the requirement of the theorem that the block selection condition contain no $in()$ atoms, the SELECT operator does not consider feature sets of its input blocks. On the other hand, by the definition, SELECT is only allowed to delete blocks, but not change their features, while PROJECT can change block features but can't delete blocks. Then let us look at an arbitrary block $b \in v$, case by case:

1. If $max_\theta(\lambda(C_2[B/b]\theta)) < p_2$ then b is deleted by the inner SELECT operator in v', and by the outer SELECT operator in v''. Thus, $b' \notin v'$ and $b'' \notin v''$.

2. If $max_\theta(\lambda(C_2[B/b]\theta)) \geq p_2$ then b is preserved by SELECT operators in both v' and v'' and passed through PROJECT . As it is the same PROJECT operator applied to the same block b, we can say that $b' = b''$ in this case.

Thus, we have shown that an arbitrary block $b \in v$ either does not have corresponding blocks in v', v'', or it has such blocks $b' \in v'$ and $b'' \in v''$ that $b' = b''$. Therefore,

$v' = v''$.

4.3.4 The APPLY Operators

There are two APPLY operators in VDA . Both serve to modify the *content* of blocks as opposed to their features or time information. To define these APPLY operators, we should first define the *transformation function*:

Definition 4.3.10 (Block Transformation Function) *A* block transformation function tr *takes a video block* b *and a region* r*, and returns a new block* $tr(b, r) = \langle b.t_s, b.t_e, b.FS, DS\rangle$*, where* DS *is such that any pixel outside of* r *is the same as in* $b.DS$*.*

The first instance of APPLY is similar to the SELECT operator but it *modifies* blocks instead of selecting them:

Definition 4.3.11 (Block APPLY Operator) *Given a video summary* v*, a block selection condition* C*, a real number* $p_{min} \in [0, 1]$*, and a block transformation function* tr*, the block* APPLY *operator* $\beta_C^{tr}(v, p_{min})$ *produces a new summary*

$$\beta_C^{tr}(v, p_{min}) = \{ \begin{cases} tr(b, \nabla) & if\ max_{all\ \theta}(\lambda(C[B/b]\theta)) \geq p_{min} \\ b & otherwise \end{cases} \mid \forall b \in v\},$$

where ∇ *represents the whole frame and* θ *is any variable substitution that grounds all remaining variables in* $C[B/b]$ *by replacing them with distinct blocks from* v*.*

The second version of APPLY is similar to the PROJECT operator but it *modifies* features instead of selecting them:

Definition 4.3.12 (Feature APPLY Operator) *Given a video summary v, a feature selection condition C, a real number $p_{min} \in [0,1]$, and a block transformation function tr, the feature* APPLY *operator $\alpha^{tr}_C(v, p_{min})$ produces a new summary*

$$\alpha^{tr}_C(v, p_{min}) = \{tr(b, \bigcup_{\{f \in b.FS | max_{all\ \theta}(\lambda(C[F/f]\theta)) \geq p_{min}\}} f.loc) \mid \forall b \in v\},$$

where θ is any variable substitution that grounds all remaining variables in $C[F/f]$ by replacing them with distinct frames from $b.FS$.

Notice that neither APPLY operator modifies block feature sets. This may cause a situation where features no longer reflect block contents correctly. If that is the case, the user may want to remove invalidated features with the PROJECT operator.

Given certain properties of their transformation functions, APPLY operators can be swapped:

Theorem 4.3.4 (Swapping APPLY Operators) *Given a video summary v, two block selection conditions C_1, C_2, two real values $p_1, p_2 \in [0,1]$, and two block transformation functions tr_1, tr_2 such that $tr_1(tr_2(b, r_2), r_1) = tr_2(tr_1(b, r_1), r_2)$ for any b, r_1, r_2, it is true that*

$$\beta^{tr_1}_{C_1}(\beta^{tr_2}_{C_2}(v, p_2), p_1) = \beta^{tr_2}_{C_2}(\beta^{tr_1}_{C_1}(v, p_1), p_2).$$

Same holds for a pair of feature APPLY *operators and a combination of block and feature* APPLY *operators.*

Proof of Theorem 4.3.4. Let us consider two queries $v' = \beta^{tr_1}_{C_1}(\beta^{tr_2}_{C_2}(v, p_2), p_1)$ and $v'' = \beta^{tr_2}_{C_2}(\beta^{tr_1}_{C_1}(v, p_1), p_2)$. As APPLY operators do not delete or add blocks, each block $b \in v$ will have two corresponding blocks $b' \in v'$ and $b'' \in v''$. Let us then look at an arbitrary block $b \in v$, case by case:

1. If $max_{all\ \theta}(\lambda(C_1[B/b]\theta)) < p_1$ and $max_{all\ \theta}(\lambda(C_2[B/b]\theta)) < p_2$ then $b' = b$ and $b'' = b$ by the definition of block APPLY , and therefore $b' = b''$.

2. If $max_{all\ \theta}(\lambda(C_1[B/b]\theta)) < p_1$ and $max_{all\ \theta}(\lambda(C_2[B/b]\theta)) \geq p_2$ then $b' = b'' = tr_2(b, \nabla)$.

3. If $max_{all\ \theta}(\lambda(C_1[B/b]\theta)) \geq p_1$ and $max_{all\ \theta}(\lambda(C_2[B/b]\theta)) < p_2$ then $b' = b'' = tr_1(b, \nabla)$.

4. Finally, if $max_{all\ \theta}(\lambda(C_1[B/b]\theta)) \geq p_1$ and $max_{all\ \theta}(\lambda(C_2[B/b]\theta)) \geq p_2$ then $b' = tr_1(tr_2(b, \nabla), \nabla)$ and $b'' = tr_2(tr_1(b, \nabla), \nabla)$. Given the theorem requirement that $tr_1(tr_2(b, r_2), r_1) = tr_2(tr_1(b, r_1), r_2)$ we can yet again say that $b' = b''$.

Thus, we have shown that $b' = b''$ for any block $b \in v$ and therefore $v' = v''$. Similar proof can be given for a pair of feature APPLY operators and a combination of feature and block APPLY operators.

As APPLY only operates on the *image data* but does not touch features and timing information, APPLY and SELECT can be swapped:

Theorem 4.3.5 (Swapping APPLY and SELECT) *Given a video summary v, two block selection conditions C_1, C_2, two real values $p_1, p_2 \in [0, 1]$, and a block transformation function tr, if C_2 is local then*

$$\sigma_{C_1}(\beta^{tr}_{C_2}(v, p_2), p_1) = \beta^{tr}_{C_2}(\sigma_{C_1}(v, p_1), p_2).$$

Same holds for a feature APPLY *operator, but* any *feature selection condition C_2 is allowed, it does not have to be local.*

Proof of Theorem 4.3.5. Let us consider two queries $v' = \sigma_{C_1}(\beta^{tr}_{C_2}(v, p_2), p_1)$ and $v'' = \beta^{tr}_{C_2}(\sigma_{C_1}(v, p_1), p_2)$. By definitions of SELECT and APPLY operators, if a block

$b \in v$ has a corresponding block $b' \in v'$ then it also has a corresponding block $b'' \in v''$ and vice versa. There are no blocks in v', v'' that have no corresponding blocks in v. Let us then look at an arbitrary block $b \in v$, case by case:

1. If $max_{all\ \theta}(\lambda(C_1[B/b]\theta)) < p_1$ and $\lambda(C_2[B/b]) < p_2$ then b is deleted by the outer SELECT operator in v', and by the inner SELECT operator in v''. Thus $b' \notin v'$ and $b'' \notin v''$.

2. If $max_{all\ \theta}(\lambda(C_1[B/b]\theta)) < p_1$ and $\lambda(C_2[B/b]) \geq p_2$ then $tr(b, \nabla)$ is deleted by the outer SELECT operator in v', while b is deleted by the inner SELECT operator in v''. Thus $b' \notin v'$ and $b'' \notin v''$.

3. If $max_{all\ \theta}(\lambda(C_1[B/b]\theta)) \geq p_1$ then b is preserved by SELECT operators in both v' and v'' and passed through APPLY . Due to C_2 being local, the APPLY operator considers b independently from all other blocks in its input and applies tr function depending on a single value of $\lambda(C_2[B/b])$. Thus, it produces identical results in both v' and v'', i.e. $b' = b''$.

We have shown that whenever b', b'' exist for an arbitrary $b \in v$, they are equal, and therefore $v' = v''$. Similar proof can be given for a combination of SELECT and feature APPLY operators, although C_2 is not required to be local in this case.

Swapping APPLY and PROJECT is also possible:

Theorem 4.3.6 (Swapping APPLY and PROJECT) *Given a video summary v, a feature selection condition C_1, a block selection condition C_2, two real values $p_1, p_2 \in [0, 1]$, and a block transformation function tr, if C_2 has no $in()$ atoms then*

$$\pi_{C_1}(\beta^{tr}_{C_2}(v, p_2), p_1) = \beta^{tr}_{C_2}(\pi_{C_1}(v, p_1), p_2).$$

Unfortunately, this theorem does not hold for feature APPLY operators.

Proof of Theorem 4.3.6. Let us consider two queries $v' = \pi_{C_1}(\beta^{tr}_{C_2}(v, p_2), p_1)$ and $v'' = \beta^{tr}_{C_2}(\pi_{C_1}(v, p_1), p_2)$. Due to definitions of APPLY and PROJECT , every block $b \in v$ will have corresponding blocks $b' \in v'$ and $b'' \in v''$. As the theorem requires C_2 to be free of $in()$ atoms, neither APPLY operator considers feature sets of its input blocks. Also, by definition, APPLY only modifies block image data. On the other hand, both PROJECT operators consider and modify input block features while ignoring the rest of the block data. In short, we can write

$$b' = b'' = \begin{cases} tr(b^*, \nabla) & if\ max_{all\ \theta}(\lambda(C_2[B/b]\theta)) \geq p_2 \\ b^* & otherwise \end{cases},$$

where $b^* = \langle b.t_s,\ b.t_e,\ \{f \in b.FS \mid\ max_{all\ \theta}(\lambda(C_1[F/f]\theta)) \geq p_1\},\ b.DS\rangle$. Thus, $b' = b''$ for every $b \in v$ and therefore $v' = v''$.

4.3.5 The MATCH Operator

When querying a video, users may often want to find video fragments that look *similar* to a supplied sample. For example, a security guard who has an image of a suspicious car approaching the gate may want to find this car in video streams coming from other cameras installed throughout the guarded area. In other instance of such a search, a TV broadcaster may want to search through the video archive looking for fragments similar to the video he has already selected for tonight's show. To support such similarity searches in the algebra, let us first see what it means for two video summaries to be similar.

In all similarity searches, there are two input videos (or video summaries in our case): the *data* that is being searched and the *pattern* that we are trying to *match* against the data. Patterns may be as small as a single frame or as long as several minutes of video. To measure similarity between two video summaries, let us introduce

the *similarity function*.

Definition 4.3.13 (Similarity Function) *Given any two video summaries v_1 and v_2, the* similarity function $sim(v_1, v_2)$ *returns a real value in the* $[0, 1]$ *range such that* $sim(v_1, v_1) = 1$.

Intuitively, the similarity function measures how similar its arguments are. Notice that the definition only requires $sim()$ to be reflexive, but neither commutativity nor transitivity are required. Notice also that the definition does not state what portions of v_1 and v_2 are considered. In practice, the $sim()$ function will often consider entire v_2 (the pattern) but only a portion of v_1 (the data), starting from the beginning. The following example illustrates how a similarity function can be computed with respect to the VDA data model.

Example 4.3.1 (Similarity Function) *For this example, let us ignore block lengths and compare two video summaries block by block. We will measure similarity between each two blocks by the spatial placement of corresponding features in these blocks:*

$$
\begin{aligned}
sim_{loc}(v_1, v_2) &= \frac{\sum\{sim_{loc}(b_i, b'_i) \mid i \in [1, min(card(v_1), card(v_2))] \wedge b_i \in v_1 \wedge b'_i \in v_2\}}{min(card(v_1), card(v_2))}, \\
sim_{loc}(b_1, b_2) &= \frac{\sum\{\frac{area(XS(b_1,N) \cap XS(b_2,N))}{area(XS(b_2,N))} \mid \forall N \in NS(b_2)\}}{card(NS)}, \\
XS(b, N) &= \bigcup\{f.loc \mid \forall f \in b.FS \, \langle f.name, f.val\rangle = N\}, \\
NS(b) &= \{\langle f.name, f.val\rangle \mid f \in b.FS\}.
\end{aligned}
$$

The $NS(b_2)$ set contains all distinct name-value pairs from the feature set of the "pattern" block b_2. The total regions occupied by each pair $N \in NS$ are then computed as $XS(b_1, N)$ (for the "data" block) and $XS(b_2, N)$ (for the "pattern" block), and the total area of their intersection is compared to the total area of $XS(b_2, N)$. Thus, the more similar $XS(b_1, N)$ and $XS(b_2, N)$ are, the higher $\frac{area(XS(b_1,N) \cap XS(b_2,N))}{area(XS(b_2,N))}$

ratio becomes. Finally, these ratios are added together for all name-value pairs and blocks, and the result is scaled to the $[0, 1]$ *range.*

Many other similarity functions are possible. For example, one may use the

$$sim'_{loc}(v_1, v_2) = \frac{sim_{loc}(v_1, v_2) + sim_{loc}(v_2, v_1)}{2}$$

function that has an additional property of commutativity.

Let us now define the MATCH operator that finds in its input k fragments that are most similar to the supplied pattern.

Definition 4.3.14 (MATCH Operator) *Consider the set* $\Sigma(v_1)$ *of all subsets of a video summary* v_1 *such that for any* $v \in \Sigma(v_1)$*, if* $b_1, b_2 \in v$ *and* $b \in v_1$ *then* $b_1 \leq b \leq b_2 \rightarrow b \in v$*. Given the second video summary* v_2 *and an integer* k*, the* MATCH *operator returns a new summary such that*

$$\mu_k(v_1, v_2) = \bigcup\{v \in \Sigma(v_1) \mid card(\{v' \in \Sigma(v_1) \mid sim(v', v_2) > sim(v, v_2)\}) < k\}.$$

Thus, the MATCH operator returns a summary containing the k fragments of the v_1 best matching the v_2.

4.3.6 The Cartesian Product Operator

While not immediately useful, the cartesian product operator serves as a basis for the JOIN operator. The idea of CPRODUCT is to take two video summaries and combine them in a way described by the *block combination function*:

Definition 4.3.15 (Block Combination Function) *Given a single block* b *and a set of blocks* $S = \{b_1, \ldots, b_n\}$*, a block combination function* g *produces a single block* $b' = g(b, S)$ *such that* $b'.t_s = b.t_s$*,* $b'.t_e = b.t_e$*, and* $g(b, \emptyset) = b$*.*

It is important to note that, while the second argument of g *may* be a summary, it does not have to be one. The blocks may come from completely different sources and overlap one another. A block combination function may use different strategies to integrate these blocks into its first argument, such as compressing video, "split-screen", and "picture-in-picture" effects.

Definition 4.3.16 (Cartesian Product Operator) *Given two video summaries v_1, v_2 and a block combination function g, the cartesian product operator $v_1 \times^g v_2$ produces a new summary*

$$v_1 \times^g v_2 = \{g(b, v_2) \mid \forall b \in v_1\}.$$

4.3.7 The JOIN Operator

The JOIN operator is similar to the relational join. What makes it different is the ability to combine video blocks using a block combination function. The JOIN is basically a version of CPRODUCT where inputs are filtered before being merged:

Definition 4.3.17 (JOIN Operator) *Given two video summaries v_1, v_2, a block join condition C, a real value $p_{min} \in [0, 1]$, and a block combination function g, the* JOIN *operator $v_1 \bowtie^g_{C,p_{min}} v_2$ produces a new summary*

$$v_1 \bowtie^g_{C,p_{min}} v_2 = \{g(b, v) \mid \forall b \in v_1 \; v = \{b' \in v_2 \mid \lambda(C[B_1/b, B_2/b']) \geq p_{min}\}\}.$$

For a JOIN example, let us consider two cameras mounted on a toll booth and directed at the driver (through the windshield) and at the license plate at the back of his car. To combine feeds from these cameras, assume the existance of a block combination function $g_{PiP}(b, S)$ that displays members of S in small boxes starting

from the left upper corner of block b. Then a query that combines videos from two cameras v_{front} and v_{back} will look like this:

$$v_{front} \bowtie^{g_{PiP}}_{B_1=\langle Car,true\rangle \wedge B_2=\langle LicensePlate,true\rangle \wedge overlap(B_1,B_2)} v_{back}.$$

A special case of the JOIN operator is the *selective join* defined as follows:

Definition 4.3.18 (Selective JOIN Operator) *Given two video summaries v_1, v_2, a block join condition C, and a real value $p_{min} \in [0,1]$, the* selective JOIN *operator $v_1 \bowtie_{C,p_{min}} v_2$ produces a new summary*

$$v_1 \bowtie_{C,p_{min}} v_2 = \{b \in v_1 \mid \exists b' \in v_2 \; \lambda(C[B_1/b, B_2/b']) \geq p_{min}\}.$$

The selective JOIN does not require a block combination function and can be used for selecting video blocks that are somehow related to the blocks in the other video. For example, to select scenes of celebration following goals in a soccer video v, one can use the following query:

$$v_{public} \bowtie_{B_1=\langle Celebration,true\rangle \wedge B_2=\langle Goal,true\rangle \wedge after(B_1,B_2,15)} v_{field}.$$

In this example, celebration scenes are taken from the camera directed at spectators (v_{public}), while the goal footage comes from another camera "watching" the playfield (v_{field}).

Theorem 4.3.7 (Swapping JOIN and SELECT) *Given two video summaries v_1, v_2, a block selection condition C_1, a block join condition C_2, a block combination function g, and two real values $p_1, p_2 \in [0,1]$, if $g(b,S).FS = b.FS$ for all b, S then*

$$\sigma_{C_1}(v_1 \bowtie^{g}_{C_2,p_2} v_2, p_1) = \sigma_{C_1}(v_1, p_1) \bowtie^{g}_{C_2,p_2} v_2.$$

Same holds for the selective JOIN , although there is no g in this case and C_1 has to be local.

Proof of Theorem 4.3.7. Consider two queries $v' = \sigma_{C_1}(v_1 \bowtie^g_{C_2,p_2} v_2, p_1)$ and $v'' = \sigma_{C_1}(v_1, p_1) \bowtie^g_{C_2,p_2} v_2$. By the block combination function definition, $g(b,S).t_s = b.t_s$ and $g(b,S).t_e = b.t_e$ for any b, S. Furthermore, the theorem requires that $g(b,S).FS = b.FS$, i.e. the only part of b that g can change is the image data $b.DS$. Thus, applying g does not affect the evaluation of the block selection condition C_1.

Consider now an arbitrary block $b \in v_1$. By definitions of JOIN and SELECT , if there is a corresponding block $b' \in v'$, there is also a block $b'' \in v''$ and vice versa. There are no blocks in v', v'' without corresponding blocks in v_1. Let us then use operator definitions to compute b', b'' for an arbitrary block $b \in v_1$, case by case:

1. If $max_{all\ \theta}(\lambda(C_1[B/b]\theta)) < p_1$ then $g(b, \{b^* \in v_2 \mid \lambda(C_2[B_1/b, B_2/b^*]) \geq p_2\})$ is deleted by the outer SELECT operator in v' while b is deleted by the inner SELECT operator in v''. Therefore, $b' \notin v'$ and $b'' \notin v''$.

2. If $max_{all\ \theta}(\lambda(C_1[B/b]\theta)) \geq p_1$ then

$$b' = b'' = g(b, \{b^* \in v_2 \mid \lambda(C_2[B_1/b, B_2/b^*]) \geq p_2\}).$$

We have shown that $b' = b''$ for any $b \in v_1$. Therefore $v' = v''$. Similar proof can be given for a combination of SELECT and selective JOIN , although C_1 has to be local in this case.

Theorem 4.3.8 (Swapping JOIN and PROJECT) *Given two video summaries v_1, v_2, a feature selection condition C_1, a block join condition C_2, a block combination function g, and two real values $p_1, p_2 \in [0,1]$, if C_2 contains no $in(B_1, -)$ atoms and $g(b,S)$ neither uses nor modifies $b.FS$ for all b, S then*

$$\pi_{C_1}(v_1 \bowtie^g_{C_2,p_2} v_2, p_1) = \pi_{C_1}(v_1, p_1) \bowtie^g_{C_2,p_2} v_2.$$

Same holds for the selective JOIN , although there is no g in this case.

Proof of Theorem 4.3.8. Consider two queries $v' = \pi_{C_1}(v_1 \bowtie^g_{C_2,p_2} v_2, p_1)$ and $v'' = \pi_{C_1}(v_1, p_1) \bowtie^g_{C_2,p_2} v_2$, and an arbitrary block $b \in v_1$. Neither JOIN nor PROJECT add or delete any blocks. Therefore, b always has two corresponding blocks $b' \in v'$ and $b'' \in v''$ and there are no blocks in v', v'' that have no corresponding blocks in v_1. By the block combination function definition, $g(b, S).t_s = b.t_s$ and $g(b, S).t_e = b.t_e$ for any b, S. Furthermore, the theorem requires that $g(b, S).FS = b.FS$, i.e. the only part of b that g can change is the image data $b.DS$. Thus, applying g does not affect the evaluation of the feature selection condition C_1. On the other hand, the theorem requires that neither evaluation of $C_2[B_1/b]$ nor the output of $g(b, S)$ depend on $b.FS$. Thus, they are not affected by PROJECT deleting features from $b.FS$. In other words

$$\begin{aligned} b' &= b'' = g(\langle b.t_s, b.t_e, FS, DS\rangle, \\ FS &= \{f \in b.FS | max_{all\ \theta}(\lambda(C_1[F/f]\theta)) \geq p_1\}, \\ DS &= g(b, \{b^* \in v_2 | \lambda(C_2[B_1/b, B_2/b^*]) \geq p_2\}).DS. \end{aligned}$$

We have shown that $b' = b''$ for any $b \in v_1$. Therefore $v' = v''$. Similar proof can be given for a combination of PROJECT and selective JOIN .

Theorem 4.3.9 (Swapping JOIN and APPLY) *Given two video summaries v_1, v_2, a block selection condition C_1, a block join condition C_2, a block transformation function tr, a block combination function g, and two real values $p_1, p_2 \in [0, 1]$,*
if $g(b, S).FS = b.FS$ and $tr(g(b, S), r) = g(tr(b, r), S)$ for all b, S, r then

$$\beta^{tr}_{C_1}(v_1 \bowtie^g_{C_2,p_2} v_2, p_1) = \beta^{tr}_{C_1}(v_1, p_1) \bowtie^g_{C_2,p_2} v_2.$$

Same equivalence holds for the feature APPLY *operator. It also holds for the selective* JOIN *, although C_1 has to be local and there is no g in this case. Finally, it holds for a combination of feature* APPLY *and selective* JOIN *, where there are no restrictions on C_1.*

Proof of Theorem 4.3.9. Consider two queries $v' = \beta^{tr}_{C_1}(v_1 \bowtie^{g}_{C_2,p_2} v_2, p_1)$ and $v'' = \beta^{tr}_{C_1}(v_1, p_1) \bowtie^{g}_{C_2,p_2} v_2$. By the block combination function definition, $g(b,S).t_s = b.t_s$ and $g(b,S).t_e = b.t_e$ for any b, S. Furthermore, the theorem requires that $g(b,S).FS = b.FS$, i.e. the only part of b that g can change is the image data $b.DS$. Thus, applying g does not affect the evaluation of C_1. On the other hand, APPLY operator only changes image data of its input blocks and therefore it does not affect the evaluation of C_2.
Consider now an arbitrary block $b \in v_1$. As neither JOIN nor APPLY add or delete any blocks, b always has corresponding blocks $b' \in v'$ and $b'' \in v''$ and there are no blocks in v', v'' that have no corresponding blocks in v_1. Let us then use operator definitions to compute b', b'' for an arbitrary block $b \in v_1$:

$$b' = \begin{cases} tr(g(b, \{b^* \in v_2 | \lambda(C_2[B_1/b, B_2/b^*]) \geq p_2\}), \nabla) & if\ max_{all\ \theta}(\lambda(C_1[B/b]\theta)) \geq p_1 \\ g(b, \{b^* \in v_2 | \lambda(C_2[B_1/b, B_2/b^*]) \geq p_2\}) & otherwise \end{cases},$$

$$b'' = \begin{cases} g(tr(b, \nabla), \{b^* \in v_2 | \lambda(C_2[B_1/b, B_2/b^*]) \geq p_2\}) & if\ max_{all\ \theta}(\lambda(C_1[B/b]\theta)) \geq p_1 \\ g(b, \{b^* \in v_2 | \lambda(C_2[B_1/b, B_2/b^*]) \geq p_2\}) & otherwise \end{cases}.$$

As the theorem requires that $tr(g(b,S), r) = g(tr(b,r), S)$, we can say that $b' = b''$ for any $b \in v_1$. Therefore $v' = v''$. Similar proof can be given for combinations of feature APPLY and selective JOIN , although in the case of block APPLY combined with selective JOIN C_1 has to be local.

4.3.8 Set Operators

Intuitively, set operators should work similarly to their canonical definitions by treating video summaries as sets of blocks, but this leads to two questions: (i) What makes two video blocks equivalent? and (ii) How to combine video blocks that overlap in time?

Let us first introduce the equivalence relation "$\sim$" defined on blocks. This relation can be chosen from a variety of possibilities. For example, we can attach IDs to all the blocks in the database and define $b_1 \sim b_2 \leftrightarrow b_1.id = b_2.id$. Or, one can say that $b_1 \sim b_2 \leftrightarrow b_1.t_s = b_2.t_s \wedge b_1.t_e = b_2.t_e$. Let us define the *set intersection and difference* operators with respect to the "$\sim$" relation:

Definition 4.3.19 (Set Intersection and Difference Operators) *Given two video summaries v_1, v_2, and an equivalence relation "$\sim$" on blocks, the set intersection and difference operators return new summaries such that*

$$\begin{aligned} v_1 \cap_\sim v_2 &= \{b \in v_1 \mid \exists b' \in v_2\ b \sim b'\}, \\ v_1 -_\sim v_2 &= \{b \in v_1 \mid \forall b' \in v_2\ b \not\sim b'\}. \end{aligned}$$

Unfortunately, the set union operator cannot be defined as easily as the intersection and difference because it requires merging overlapping video blocks into new blocks. To deal with this problem, let us define *split* and *merge* operators:

Definition 4.3.20 (Split Operator) *Given two video summaries v_1, v_2, the* split *function $split(v_1, v_2)$ returns a new video summary such that*

$$\begin{aligned} S(v) &= \{t \mid \forall b \in v\ t = b.t_s \vee t = b.t_e\}, \\ split(v_1, v_2) &= \bigcup_{b \in v_1} \{\langle t_s, t_e, b.FS, DS\rangle \mid \forall t_s, t_e \in S(v_2) \cup \{b.t_s, b.t_e\}\ [t_s, t_e] \subseteq [b.t_s, b.t_e]\}, \end{aligned}$$

where each DS consists of the $[t_s, t_e)$ range of frames taken from $b.DS$.

Intuitively, the $split(v_1, v_2)$ operator splits v_1 blocks at boundaries imposed by blocks from v_2, so that each resulting block overlaps with *at most one* block from v_2 and there are no partial overlaps. To merge overlapping blocks together, we will need one more operator:

Definition 4.3.21 (Merge Operator) *Given two video summaries v_1, v_2 and a block combination function g, the* merge *operator $merge^g(v_1, v_2)$ returns a new video summary such that*

$$\begin{aligned} merge^g(v_1, v_2) &= \{g(b, v') \mid \forall b \in v_1\ v' = \{b' \in v_2 \mid [b.t_s, b.t_e) \cap [b'.t_s, b'.t_e) \neq \emptyset\}\} \\ &\cup \{b \in v_1 \mid \forall b' \in v_2\ [b.t_s, b.t_e) \cap [b'.t_s, b'.t_e) = \emptyset\} \\ &\cup \{b \in v_2 \mid \forall b' \in v_1\ [b.t_s, b.t_e) \cap [b'.t_s, b'.t_e) = \emptyset\}. \end{aligned}$$

By using the $split()$ and $merge()$ operators, one can define the *set union* in a following way:

Definition 4.3.22 (Set Union Operator) *Given two video summaries v_1, v_2, an equivalence relation "$\sim$" defined on blocks, and a block combination function g, the* set union *operator produces a new summary such that*

$$\begin{aligned} v' &= \{b \in v_2 \mid \forall b' \in v_1\, b' \not\sim b\}, \\ v_1 \cup^g_\sim v_2 &= merge^g(split(v_1, v'), split(v', v_1)). \end{aligned}$$

The above definition of set operators allows for many useful equivalences. Let us see some of them.

Theorem 4.3.10 (Swapping Set Operators and SELECT) *Given two video summaries v_1, v_2, a block selection condition C, a real value $p \in [0, 1]$, and an equivalence relation "$\sim$", if C is local then*

$$\begin{aligned} \sigma_C(v_1, p) \cap_\sim v_2 &= \sigma_C(v_1 \cap_\sim v_2, p), \\ \sigma_C(v_1, p) -_\sim v_2 &= \sigma_C(v_1 - \sim v_2, p). \end{aligned}$$

Proof of Theorem 4.3.10. Consider two queries $v' = \sigma_C(v_1, p) \cap_\sim v_2$ and $v'' = \sigma_C(v_1 \cap_\sim v_2, p)$. As the theorem requires C to be local, both SELECT operators consider each input block independently of other blocks. By definitions of SELECT and set intersection, if a block $b \in v_1$ has a corresponding block $b' \in v'$, it must also have a corresponding block $b'' \in v''$ and vice versa. There are no blocks in v', v'' that have no corresponding blocks in v_1. Let us then consider an arbitrary block $b \in v_1$ and use operator definitions to compute its corresponding blocks b', b'', case by case:

1. If $\lambda(C[B/b]) < p$ then b is deleted by the inner SELECT operator in v' and by the outer SELECT operator in v''. Thus, $b' \notin v'$ and $b'' \notin v''$.

2. If $\lambda(C[B/b]) \geq p$ then $b' = b'' = b$ if and only if there is $b^* \in v_2$ such that $b \sim b^*$. Otherwise, $b' \notin v'$ and $b'' \notin v''$.

We have shown that b', b'' either do not exist or $b' = b''$, for any $b \in v_1$. Therefore, $v' = v''$. Similar proof can be given for a combination of SELECT and set difference.

Theorem 4.3.11 (Swapping Set Operators and APPLY) *Given two video summaries v_1, v_2, a block selection condition C, a real value $p \in [0, 1]$, a transformation function tr, and an equivalence relation "$\sim$" such that $b_1 \sim b_2 \leftrightarrow tr(b_1, r) \sim b_2$ for any b_1, b_2, r, if C is local then*

$$\begin{aligned} \beta_C^{tr}(v_1, p) \cap_\sim v_2 &= \beta_C^{tr}(v_1 \cap_\sim v_2, p), \\ \beta_C^{tr}(v_1, p) -_\sim v_2 &= \beta_C^{tr}(v_1 - \sim v_2, p). \end{aligned}$$

Same holds for a feature APPLY *operator, although C does not have to be local in this case.*

Proof of Theorem 4.3.11. Consider two queries $v' = \beta_C^{tr}(v_1, p) \cap_\sim v_2$ and $v'' = \beta_C^{tr}(v_1 \cap_\sim v_2, p)$. As the theorem requires C to be local, both APPLY operators consider each input block independently of other blocks. By definitions of APPLY and set intersection, if a block $b \in v_1$ has a corresponding block $b' \in v'$, it also has a corresponding block $b'' \in v''$ and vice versa. There are no blocks in v', v'' that have no corresponding blocks in v_1. Let us then look at an arbitrary block $b \in v_1$ and use operator definitions to compute its corresponding blocks b', b'', case by case:

1. If $\lambda(C[B/b]) < p$ then neither APPLY operator modifies b and therefore $b' = b'' = b$ if and only if there is $b^* \in v_2$ such that $b \sim b^*$. Otherwise, $b' \notin v'$ and $b'' \notin v''$.

2. If $\lambda(C[B/b]) \geq p$ then the set intersection operator in v' receives as input a block $tr(b, \nabla)$ while the set intersection operator in v'' receives b^* as input. The theorem requires that $b \sim b^* \leftrightarrow tr(b, r) \sim b^*$ and therefore both input blocks will either pass set intersection or be deleted by it. Thus, the first query will generate $b' = tr(b, \nabla)$ while the second query applies tr and also generates $b'' = tr(b, \nabla)$, i.e. $b' = b''$.

We have shown that whenever b', b'' exist, it is true that $b' = b''$, for any $b \in v_1$. Therefore, $v' = v''$. Similar proof can be given for combinations of feature APPLY and set difference, although C does not have to be local in feature APPLY .

4.3.9 The Concatenation Operator

A simple but common video processing task is to "glue" two or more summaries together. Here is an algebraic operator to do it:

Definition 4.3.23 (Concatenation Operator) *Given two video summaries v_1, v_2 the concatenation operator $v_1 \oplus v_2$ produces a new summary*

$$v_1 \oplus v_2 = v_1 \cup \{\langle b.t_s + t_0, b.t_e + t_0, b.FS, b.DS\rangle \mid \forall v \in v_2\},$$

where $t_0 = max_{b \in v_1} b.t_e$.

Two concatenations can be easily swapped.

Theorem 4.3.12 (Swapping Concatenations) *Suppose v_1, v_2, v_3 are summaries. Then*

$$(v_1 \oplus v_2) \oplus v_3 = v_1 \oplus (v_2 \oplus v_3).$$

Proof of Theorem 4.3.12. Consider an arbitrary block b from one of v_1, v_2, v_3 summaries and two new summaries $v' = (v_1 \oplus v_2) \oplus v_3$ and $v'' = v_1 \oplus (v_2 \oplus v_3)$. As concatenation operator neither deletes nor adds blocks, b will always have counterparts $b' \in v'$ and $b'' \in v''$, and there are no blocks in v', v'' that do not have a corresponding block b. We can then write the following correspondence between these blocks:

$$
\begin{aligned}
t_1 &= max_{b \in v_1} b.t_e \\
t_2 &= max_{b \in v_2} b.t_e \\
b'.FS &= b''.FS = b.FS \\
b'.DS &= b''.DS = b.DS \\
b'.t_s &= \begin{cases} b.t_s & if\ b \in v_1 \\ b.t_s + t_1 & if\ b \in v_2 \\ b.t_s + t_1 + t_2 & if b \in v_3 \end{cases} \\
b''.t_s &= \begin{cases} b.t_s & if\ b \in v_1 \\ b.t_s + t_1 & if\ b \in v_2 \\ b.t_s + t_2 + t_1 & if b \in v_3 \end{cases} \\
b'.t_e &= \begin{cases} b.t_e & if\ b \in v_1 \\ b.t_e + t_1 & if\ b \in v_2 \\ b.t_e + t_1 + t_2 & if b \in v_3 \end{cases} \\
b''.t_e &= \begin{cases} b.t_e & if\ b \in v_1 \\ b.t_e + t_1 & if\ b \in v_2 \\ b.t_e + t_2 + t_1 & if b \in v_3 \end{cases}
\end{aligned}
$$

Thus, $b' = b''$ for any input block b and therefore $v' = v''$.

Concatenation can also be swapped with PROJECT and APPLY operators.

Theorem 4.3.13 (Swapping Concatenation and PROJECT) *Suppose v_1 and v_2 are summaries, C is a feature selection condition, and $p \in [0,1]$ is a real number. Then*

$$\pi_C(v_1, p) \oplus \pi_C(v_2, p) = \pi_C(v_1 \oplus v_2, p).$$

Proof of Theorem 4.3.13. Consider two summaries $v' = \pi_C(v_1, p) \oplus \pi_C(v_2, p)$ and $v'' = \pi_C(v_1 \oplus v_2, p)$ and an arbitrary block b in v_1, v_2. As PROJECT and concatenation operators neither delete nor add any blocks, b will always have two corresponding blocks $b' \in v'$ and $b'' \in v''$. Furthermore, the PROJECT operator only considers and modifies feature sets of input blocks, while concatenation operator only considers and modifies t_s, t_e values of input blocks. In other words:

$$\begin{aligned}
t_1 &= max_{b \in v_1} b.t_e \\
b'.FS &= b''.FS = \{f \in b.FS \mid max_{all\ \theta}(\lambda(C[F/f]\theta)) \geq p\} \\
b'.DS &= b''.DS = b.DS \\
b'.t_s &= \begin{cases} b.t_s & if\ b \in v_1 \\ b.t_s + t_1 & if\ b \in v_2 \end{cases} \\
b''.t_s &= \begin{cases} b.t_s & if\ b \in v_1 \\ b.t_s + t_1 & if\ b \in v_2 \end{cases} \\
b'.t_e &= \begin{cases} b.t_e & if\ b \in v_1 \\ b.t_e + t_1 & if\ b \in v_2 \end{cases} \\
b''.t_e &= \begin{cases} b.t_e & if\ b \in v_1 \\ b.t_e + t_1 & if\ b \in v_2 \end{cases}
\end{aligned}$$

Thus, $b' = b''$ for any input block b and therefore $v' = v''$.

Theorem 4.3.14 (Swapping Concatenation and APPLY) *Suppose v_1 and v_2 are summaries, C is a local block condition, tr is a block transformation function, and*

$p \in [0,1]$ *is a real number. Then*

$$\beta_C^{tr}(v_1, p) \oplus \beta_C^{tr}(v_2, p) = \beta_C^{tr}(v_1 \oplus v_2, p).$$

Same holds for feature APPLY *operators, although C does not have to be local in this case.*

Proof of Theorem 4.3.14. Consider two queries $v' = \beta_C^{tr}(v_1, p) \oplus \beta_C^{tr}(v_2, p)$ and $v'' = \beta_C^{tr}(v_1 \oplus v_2, p)$, and an arbitrary block b in v_1, v_2. As APPLY and concatenation operators neither delete nor add any blocks, b will always have two corresponding blocks $b' \in v'$ and $b'' \in v''$. Furthermore, due to locality of C, the APPLY operator considers its input one block at a time, ignoring t_s, t_e values and only modifying image data, while concatenation operator only considers and modifies t_s, t_e values of input blocks. In other words:

$$\begin{aligned}
t_1 &= max_{b \in v_1} b.t_e, \\
b^* &= \langle \begin{cases} b.t_s + t_1 & if\ b \in v_2 \\ b.t_s & otherwise \end{cases}, \begin{cases} b.t_e + t_1 & if\ b \in v_2 \\ b.t_e & otherwise \end{cases}, b.FS, b.DS \rangle, \\
b' &= b'' = \begin{cases} tr(b^*, \nabla) & if\ \lambda(C[B/b^*]) \geq p \\ b^* & otherwise \end{cases}
\end{aligned}$$

Thus, $b' = b''$ for any input block b and therefore $v' = v''$. Similar proof can be given for a concatenation of feature APPLY operators, although C is not required to be local in this case.

4.3.10 The COMPRESS Operator

By the definition of the video summary, it may contain "gaps" between blocks, where content is missing or removed as result of queries. A video browsing and display

application may choose to hide these gaps from the viewer by skipping directly to the next block as the current one finishes playing. To permanently remove gaps from a summary though, one will need an operator to *compress* summaries:

Definition 4.3.24 (COMPRESS **Operator)** *Given a summary v, the* COMPRESS *operator $\eta(v)$ produces a new summary*

$$\eta(v) = \{\langle t_s, t_s + b.t_e - b.t_s, b.FS, b.DS\rangle \mid \forall b \in v\ t_s = \sum_{\{b' \in v | b'.t_e \leq b.t_s\}} (b'.t_e - b'.t_s)\}.$$

It is easy to notice that applying compression to a summary effectively turns it into a *video* (i.e. summary with no gaps), while applying it again will not change this video. Thus, two COMPRESS operators can be reduced to one:

Theorem 4.3.15 (Reducing COMPRESS **)** *Suppose v is a summary. Then*

$$\eta(\eta(v)) = \eta(v).$$

Proof of Theorem 4.3.15. Proving this theorem by induction:

1. Let us start with an empty summary: $\eta(\eta(\emptyset)) = \emptyset$ and $\eta(\emptyset) = \emptyset$, i.e. the theorem holds.

2. Let us denote $\eta(\eta(v))$ with v', $\eta(v)$ with v'', and assume that the theorem holds for some summary v, i.e. $v' = v''$.

3. If a block b is added to the end of v, we can use the the COMPRESS operator definition to write that

$$\begin{aligned} \eta(\eta(v \cup \{b\})) &= v' \cup \{\langle b.t_s + length(v'), b.t_e + length(v'), b.FS, b.DS\rangle\}, \\ \eta(v \cup \{b\}) &= v'' \cup \{\langle b.t_s + length(v''), b.t_e + length(v''), b.FS, b.DS\rangle\}. \end{aligned}$$

   Given assumption made in (2), we can conclude that the theorem holds for $v \cup \{b\}$.

Thus, $\eta(\eta(v)) = \eta(v)$ for any arbitrary summary v, by induction.

COMPRESS operators can be swapped with PROJECT , APPLY , and concatenation operators:

Theorem 4.3.16 (Swapping COMPRESS and PROJECT) *Suppose v is a summary, C is a feature selection condition, and $p \in [0, 1]$ is a real number. Then*

$$\pi_C(\eta(v), p) = \eta(\pi_C(v, p)).$$

Proof of Theorem 4.3.16. Proving this theorem by induction:

1. Let us start with an empty summary: $\eta(\pi_C(\emptyset, p)) = \emptyset$ and $\pi_C(\eta(\emptyset), p) = \emptyset$, i.e. the theorem holds.

2. Let us denote $\pi_C(\eta(v), p)$ with v', $\eta(\pi_C(v, p))$ with v'', and assume that the theorem holds for some summary v, i.e. $v' = v''$.

3. If a block b is added to the end of v, we can use definitions of PROJECT and COMPRESS operators to write that

$$\begin{aligned} \pi_C(\eta(v \cup \{b\}), p) &= v' \cup \{\langle b.t_s + length(v'), b.t_e + length(v'), FS, b.DS\rangle\}, \\ \eta(\pi_C(v \cup \{b\}, p)) &= v'' \cup \{\langle b.t_s + length(v''), b.t_e + length(v''), FS, b.DS\rangle\}, \\ FS &= \{f \in b.FS | max_{all\ \theta} \lambda(C[F/f]\theta)) \geq p\}. \end{aligned}$$

   Given assumption made in (2), we can conclude that the theorem holds for $v \cup \{b\}$.

Thus, $\eta(\pi_C(v, p)) = \pi_C(\eta(v), p)$ for any arbitrary summary v, by induction.

Theorem 4.3.17 (Swapping COMPRESS and APPLY) *Suppose v is a summary, C is a local block selection condition, tr is a block transformation function, and $p \in [0, 1]$ is a real number. Then*

$$\beta_C^{tr}(\eta(v)) = \eta(\beta_C^{tr}(v)).$$

Same holds for a feature APPLY *operator, where C is* any *feature condition.*

Proof of Theorem 4.3.17. Proving this theorem by induction:

1. Let us start with an empty summary: $\eta(\beta_C^{tr}(\emptyset)) = \emptyset$ and $\beta_C^{tr}(\eta(\emptyset)) = \emptyset$, i.e. the theorem holds.

2. Let us denote $\beta_C^{tr}(\eta(v))$ with v', $\eta(\beta_C^{tr}(v))$ with v'', and assume that the theorem holds for some summary v, i.e. $v' = v''$.

3. If a block b is added to the end of v, we can use definitions of APPLY and COMPRESS operators and the requirement for C to be local to write that

$$
\begin{aligned}
b'_* &= \langle b.t_s + length(v'), b.t_e + length(v'), b.FS, b.DS\rangle,\\
b''_* &= \langle b.t_s + length(v''), b.t_e + length(v''), b.FS, b.DS\rangle,\\
\beta_C^{tr}(\eta(v \cup \{b\})) &= v' \cup \{ \begin{cases} tr(b'_*, \nabla) & if\ \lambda(C[B/b'_*]) \geq p \\ b'_* & otherwise \end{cases} \},\\
\eta(\beta_C^{tr}(v \cup \{b\})) &= v' \cup \{ \begin{cases} tr(b''_*, \nabla) & if\ \lambda(C[B/b''_*]) \geq p \\ b''_* & otherwise \end{cases} \}.
\end{aligned}
$$

   Given assumption made in (2), we can conclude that the theorem holds for $v \cup \{b\}$.

Thus, $\eta(\beta_C^{tr}(v)) = \beta_C^{tr}(\eta(v))$ for any arbitrary summary v, by induction.

Theorem 4.3.18 (Swapping COMPRESS and concatenation) *Suppose v_1 and v_2 are summaries. Then*

$$\eta(v_1) \oplus \eta(v_2) = \eta(v_1 \oplus v_2).$$

Proof of Theorem 4.3.18. Consider two summaries $v' = \eta(v_1) \oplus \eta(v_2)$ and $v'' = \eta(v_1 \oplus v_2)$. As COMPRESS and concatenation operators neither delete nor add any blocks, an arbitrary block b from v_1, v_2 will have two corresponding blocks $b' \in v'$ and $b'' \in v''$, and there are no blocks in v', v'' that do not have a corresponding block b. Let us then consider two possible cases:

1. If $b \in v_1$ then, by the definition, concatenation does not modify b or put any blocks in front of it, while COMPRESS will change b so that

$$\begin{aligned} b'.t_s &= b''.t_s = \sum_{\{b^* \in v_1 | b^*.t_e \leq b.t_s\}} (b^*.t_e - b^*.t_s) \\ b'.t_e &= b''.t_e = b'.t_s + b.t_e - b.t_s \end{aligned}$$

   Thus, $b' = b''$.

2. If $b \in v_2$ then, by definitions of COMPRESS and concatenation,

$$\begin{aligned} b'.t_s &= b''.t_s = length(v_1) + \sum_{\{b^* \in v_2 | b^*.t_e \leq b.t_s\}} (b^*.t_e - b^*.t_s) \\ b'.t_e &= b''.t_e = b'.t_s + b.t_e - b.t_s \end{aligned}$$

   Again, $b' = b''$.

Thus, $b' = b''$ for any block b from either of query inputs and therefore $v' = v''$.

4.4 Indexing and Optimization

In this section, we will discuss optimizing such time-consuming operations as SELECT , JOIN , and MATCH , both by closely analyzing selection conditions and by using indices.

4.4.1 Optimizing SELECT and JOIN

As VDA mostly operates on video *descriptions* as opposed to the actual video *data*, it is quite feasible to execute most of its operations in memory, accessing disk storage only when a query needs to change the video itself (as result of APPLY or JOIN operators, for example). Consider, for example, a one-hour video divided into one-second

blocks. If each block contains an average of 10 features, we would have to keep in memory 3600 blocks and 36000 features – not a very large amount of data, by any count.

While the modest memory requirement can be considered good news, there is also bad news and it is about the way VDA treats selection conditions. Remember that such operators as SELECT and PROJECT use the *maximal value* of λ over *all possible variable assignments*. This brings the complexity of a SELECT operator with M variables in its selection condition, processing N input blocks, to $O(\frac{N!}{(N-M)!})$. This value is linear w.r.t. the size of input as long as there is only one variable, but it quickly mushrooms as the number of variables grows.

Fortunately, there is an extra consideration to make. When performing selection, a user is unlikely to specify blocks that are completely unrelated. For example, while the "give me all celebration shots in a soccer video if there are any goals in that video" request is not sensible, the "give me all celebration shots that occur in 15 second intervals after goal shots" request makes much more sense. In other words, of

$$\sigma_{in(B,name(F,\text{“}Celebration\text{”}))\wedge in(B_1,name(F,\text{“}Goal\text{”}))}(v) \qquad (1)$$

$$\sigma_{in(B,name(F,\text{“}Celebration\text{”}))\wedge in(B_1,name(F,\text{“}Goal\text{”}))\wedge after(B,B_1,15)}(v) \qquad (2)$$

queries, the second one is more likely to be issued than the first. Thus, most or all variable assignments will be *constrained in time* with respect to B. When SELECT scans its input by sequential assignment of input blocks to B, we can compute possible time ranges for all the other variables based on the time constraints and only consider assignments from within these ranges. If each time range contains an average of $K << N$ blocks, the complexity is reduced to $O(N \cdot K^{M-1})$, at the cost of time needed to compute ranges.

Consider a condition C containing variables $B, B_1, \ldots, B_m$. Given the assignment $B = b$, our task is to compute a set of constraints $\{\langle ll_i, lu_i, ul_i, uu_i\rangle \mid \forall i \in [1,m]$ $\forall\theta\ \forall b'\ \ b'.t_s \notin [ll_i, lu_i] \vee\ b'.t_e \notin [ul_i, uu_i] \rightarrow \lambda(C[B/b, B_i/b']\theta) = 0\}$. This can be done with the following algorithm:

Algorithm Ranges(C,b)
  C is a block selection condition with variables $B, B_1, \ldots, B_m$
  b is a block to be assigned to B
begin
  $RR[0] := \langle b.t_s, b.t_s, b.t_e, b.t_e\rangle$
  for $j \in [1,m]$ do
    $RR[j] := \langle 0, \infty, 0, \infty\rangle$
  end for
  do
    $i := 0$
    for $j \in [1,m]$ do
      $R := RecRanges(C, j, RR)$
      if $R \neq RR[j]$ then $i := i + 1$
      $RR[j] := R$
    end for
  while $i > 0$
  return RR
end

The $Ranges()$ algorithm starts by assigning longest possible ranges to all variables except B. It then tries to shrink these ranges one by one, by calling the recursive $RecRange()$ algorithm given below. The execution of $Ranges()$ stops when ranges cease changing. The $Ranges()$ algorithm has $O((M-1)^2)$ time complexity.

Algorithm RecRanges(C,j,RR)
  C is a block selection condition with variables $B, B_1, \ldots, B_M$
  j is the number of a variable B_j whose range we compute
  RR is the current range table
begin
  if $C = C_1 \wedge C_2$ then
    $R_1 := RecRanges(C_1, j, RR)$
    $R_2 := RecRanges(C_2, j, RR)$
    return $\langle max(R_1.ll, R_2.ll), min(R_1.lu, R_2.lu), max(R_1, ul, R_2, ul), min(R_1.uu, R_2.uu)\rangle$
  else if $C = C_1 \vee C_2$ then
    $R_1 := RecRanges(C_1, j, RR)$
    $R_2 := RecRanges(C_2, j, RR)$
    return $\langle min(R_1.ll, R_2.ll), max(R_1.lu, R_2.lu), min(R_1, ul, R_2, ul), max(R_1.uu, R_2.uu)\rangle$
  else if $C = before(B_j, B_i, d)$ then
    return $\langle 0, RR[i].lu, max(0, RR[i].ll - d), RR[i].lu\rangle$
  else if $C = before(B_i, B_j, d)$ then

```
        return ⟨RR[i].ul, min(∞, RR[i] + d), RR[i].ul, ∞⟩
    else if C = after(B_j, B_i, d) then
        return ⟨RR[i].ul, min(∞, RR[i].uu + d), RR[i].ul, ∞⟩
    else if C = after(B_i, B_j, d) then
        return ⟨0, RR[i].lu, max(0, RR[i].ll − d), RR[i].lu⟩
    else if C = within(B_j, B_i, d) or C = within(B_i, B_j, d) then
        return ⟨0, min(∞, RR[i].uu + d, max(0, RR[i].ll − d), ∞⟩
    else if C = overlap(B_j, B_i) or C = overlap(B_i, B_j) then
        return ⟨0, RR[i].uu, RR[i].ll, ∞⟩
    else if C = inside(B_j, B_i) then
        return ⟨RR[i].ll, RR[i].uu, RR[i].ll, RR[i].uu⟩
    else if C = inside(B_i, B_j) then
        return ⟨0, RR[i].ll, RR[i].uu, ∞⟩
    end if
    return ⟨0, ∞, 0, ∞⟩
end
```

When considering assignment for a variable B_i, the λ computation algorithm finds the first input block b_s, such that $b_s.t_s \geq ll_i$ and $b_s.t_e \geq ul_i$. As a video summary is an *ordered* set of blocks, the b_s block can be easily found with binary search in $O(log_2(N))$ time. The algorithm then tries assigning to B_i every block starting with b_s and ending with b_e, such that $b_e.t_s > lu_i$ or $b_e.t_e > uu_i$.

4.4.2 Optimizing MATCH

Another costly algebraic operation is *matching*. For instance, to match a pattern of size M against a video summary of size N, using the similarity measure described in Example 4.3.13, one needs to make $O((N - M) \cdot M)$ block comparisons, and each block comparison takes quadratic time with respect to the number of features in compared blocks. For large videos whose blocks contain a lot of features the MATCH operator can take quite a lot of time to execute.

To determine how to accelerate the matching process, let us first look at a "naive" matching algorithm using the $sim_{loc}()$ similarity function from Example 4.3.13:

```
Algorithm Match(v_d,v_p,k)
    v_d is the data video summary
```

v_p is the pattern video summary
k is the number of best matches to return
begin
// Priority queue is empty for now
$Q := \emptyset$

// Scan v_d
for each $j \in [1, card(v_d) - card(v_p)]$ do
$d := 0$
for each $i \in [1, card(v_p)]$ do
$d := d + MatchBlocks(v_d.b_{j+i}, v_p.b_i)$
if $card(Q) = k$ and $d + card(v_p) - i < worst(Q)$ then break
end for
if $card(Q) < k$ or $d \geq worst(Q)$ then $add(Q, \langle j, d \rangle)$
if $card(Q) > k$ then $delete(Q, last(Q))$
end for

// Return blocks corresponding to queued fragments
return $\{b_i \in v_d \mid \exists \langle j, d \rangle \in Q \; i \in [j, j + card(v_p))\}$
end

The $Match()$ algorithm scans v_d while keeping track of k best pattern matches in a sorted list Q. The algorithm terminates matching process every time it determines that the similarity value d is not going to reach the threshold needed for inclusion into Q. Aside from this trivial optimization, $Match()$ acts naively. To compute block similarity, $Match()$ uses the following algorithm:

Algorithm MatchBlocks(b_d,b_p)
b_d is the data block
b_p is the pattern block
begin
$Over := 0$
$Total := 0$
for each $f_p \in b_p.FS$ do
$Total := Total + area(f_p.loc)$
for each $f_d \in b_d.FS$ such that $f_p.name = f_d.name$ and $f_p.val = f_d.val$ do
$Over := Over + area(f_p.loc \cap f_d.loc)$
end for
end for
return $Over/Total$
end

The $MatchBlocks()$ algorithm computes the ratio of the area where b_d and b_p have same features to the total area of b_p features. It is assumed that features with

the same names and values do not overlap inside feature sets. As seen above, this algorithm has time complexity of $O(card(b_d.FS) \cdot card(b_p.FS))$. Assuming that each block contains an average of N features and the pattern is much shorter than the data, and ignoring the cost of maintaining the list Q, one can say that the total time complexity of the $Match()$ algorithm is $O(card(v_d) \cdot card(v_p) \cdot N^2)$.

One way to reduce $Match()$ complexity is by ordering features inside feature sets by their names and values. Then a binary search can be used in $MatchBlocks()$ to search for b_d features, bringing the total time complexity to $O(card(v_d) \cdot card(v_p) \cdot N \cdot log_2(N))$. Augmenting feature sets with feature name hashes can lower this figure even further.

So far, we discussed accelerating $MatchBlocks()$ but left $Match()$ algorithm untouched. To optimize $Match()$, let us first attach two additional data fields to each block b:

- $b.names$ is a bit mask whose bits correspond to all known feature names. For example, `bit0` corresponds to feature `RGB`, `bit1` corresponds to feature `MOTION`, and so forth. In practice, the number of different features is relatively small, so it may be possible to use a single `32bit` integer to hold the $names$ field.

- $b.bbox$ is the smallest rectangular region containing all features in $b.FS$. In other words, $bbox$ is the *bounding box* for all block features.

Both $b.names$ and $b.bbox$ can be computed in time linear to the number of features in $b.FS$. Looking back at the $Match()$ algorithm with respect to these new fields, one can observe the following:

1. if $b_d.names \cap b_p.names = \emptyset$ then $MatchBlocks(b_d, b_p) = 0$,

2. if $b_d.bbox \cap b_p.bbox = \emptyset$ then $MatchBlocks(b_d, b_p) = 0$,

3. $MatchBlocks(b_d, b_p) \leq \frac{area(b_d.bbox \cap b_p.bbox)}{\sum_{f \in b_p.FS} area(f.loc)}$.

The $Match()$ algorithm can now be rewritten to use the first two of these observations and avoid calling $MatchBlocks()$ for pairs of blocks that have non-overlapping feature sets. I will call the resulting algorithm $Match^+()$:

```
Algorithm Match+(v_d,v_p,k)
    v_d is the data video summary
    v_p is the pattern video summary
    k is the number of best matches to return
begin
    // Priority queue is empty for now
    Q := ∅

    // Scan v_d
    for each j ∈ [1, card(v_d) − card(v_p)] do
        d := 0
        for each i ∈ [1, card(v_p)] do
            if v_d.b_{j+i}.bbox ∩ v_p.b_i.bbox ≠ ∅ and v_d.b_{j+i}.names ∩ v_p.b_i.names ≠ ∅ then
                d := d + MatchBlocks(v_d.b_{j+i}, v_p.b_i)
            end if
        end for
        if card(Q) < k or d ≥ worst(Q) then add(Q, ⟨j, d⟩)
        if card(Q) > k then delete(Q, last(Q))
    end for

    // Fill the rest of the queue
    while card(Q) < k and exists j ∈ [1, card(v_d)] such that ⟨j, −⟩ ∉ Q do add(Q, ⟨j, 0⟩)

    // Return blocks corresponding to queued fragments
    return {b_i ∈ v_d | ∃⟨j, d⟩ ∈ Q i ∈ [j, j + card(v_p))}
end
```

4.4.3 Indexing

Even with improvements described above, the $Match()$ algorithm still needs to scan through the entire v_d. To avoid this scan, let us consider storing $bbox$ fields of all video blocks in an *R-tree* index, augmenting each tree node with a block number and a $names$ field, as follows.

Definition 4.4.1 (Augmented R-Tree Node) *Given a maximal branching factor m, an augmented R-tree node is a set of m tuples*

$$\{\langle names_1, bbox_1, c_1\rangle, \ldots, \langle names_m, bbox_m, c_m\rangle\}$$

such that for every leaf $\langle names_i, bbox_i, c_i\rangle$: (i) c_i is a corresponding block number, (ii) $names_i = b_{c_i}.names$, (iii) $bbox_i = b_{c_i}.bbox$. For every non-leaf $\langle names_i, bbox_i, c_i\rangle$: (i) c_i points to a child node or NIL, (ii) $names_i = \bigcup\{c_i.names_j | 1 \leq j \leq m \wedge c_i.c_j \neq NIL\}$, and (iii) $bbox_i = bbox(\{c_i.bbox_j | j \in [1,m] \wedge c_i.c_j \neq NIL\})$.

Augmented R-tree nodes can be used to build an R-tree for a video summary v_d by one of traditional methods, taking $names$ fields into account as an additional criterion when selecting insertion points for new leaves.

Additionally, every block's feature set can be broken into clusters, each occupying a distinct spot in a frame. Each such cluster can then be inserted into the tree *separately*, all marked with the same block number but having different $names$ fields.

Let us now rewrite the $Match^{+}()$ algorithm to make use of this tree.

```
Algorithm TreeMatch(v_d,v_p,k,T_d)
   v_d is the data video summary
   v_p is the pattern video summary
   k is the number of best matches to return
   T_d is the R-tree representing v_d
begin
   // Both priority queue and the set of candidates are empty
   Q := ∅
   C := ∅

   // Use R-tree to find candidate fragments
   for each i ∈ [1, card(v_p)] do
      TreeFind(root(T_d), C, v_p.b_i.bbox, v_p.b_i.names, i − 1)
   end for

   // Scan found v_d fragments
   for each j ∈ C such that j ∈ [1, card(v_d) − card(v_p) + 1] do
      d := 0
      for each i ∈ [1, card(v_p)] do
         if v_d.b_{j+i−1}.bbox ∩ v_p.b_i.bbox ≠ ∅ and v_d.b_{j+i−1}.names ∩ v_p.b_i.names ≠ ∅ then
```

$d := d + MatchBlocks(v_d.b_{j+i-1}, v_p.b_i)$
end if
if $card(Q) = k$ and $d + card(v_p) - i < worst(Q)$ then break
end for
if $card(Q) < k$ or $d \geq worst(Q)$ then $add(Q, \langle j, d \rangle)$
if $card(Q) > k$ then $delete(Q, last(Q))$
end for

// Fill the rest of the queue
while $card(Q) < k$ and exists $j \in [1, card(v_d)]$ such that $\langle j, - \rangle \notin Q$ do $add(Q, \langle j, 0 \rangle)$

// Return blocks corresponding to queued fragments
return $\{b_i \in v_d \mid \exists \langle j, d \rangle \in Q \; i \in [j, j + card(v_p))\}$
end

Instead of scanning through the entire v_d summary, $TreeMatch()$ calls the recursive $TreeFind()$ algorithm (shown below) to find all blocks in v_d that have non-zero similarity to at least one block in v_p and restricts matching to v_d fragments containing these blocks. While the $TreeMatch()$ worst-case complexity is higher than that of $Match^{+}()$ because of the need to search the R-tree, the *typical* execution time of the $TreeMatch()$ algorithm is dramatically better due to the greatly reduced number of fragments to match.

Algorithm TreeFind(N,C,b,l)
N is the root of the augmented R-tree
C is the set of answers
b is the pattern video block to look for
l is the offset of b inside a pattern video
begin
for each $\langle names_i, bbox_i, c_i \rangle \in N$ such that $c_i \neq NIL$ do
if $names_i \cap b.names \neq \emptyset$ and $bbox_i \cap b.bbox \neq \emptyset$ then
if $\langle names_i, bbox_i, c_i \rangle$ is leaf then $add(C, c_i - l)$ else $TreeFind(c_i, C, b, l)$
end if
end for
end

4.4.4 Experimental Results

To assess how optimization and indexing affect algebraic engine performance, I have run two batches of experiments. These experiments were conducted on a 2GHz Pen-

tium4 computer running RedHat Linux using soccer and military aircraft videos processed with a simple feature extraction tool. The feature extraction algorithm detected and classified blobs of uniform color that were later used as features.

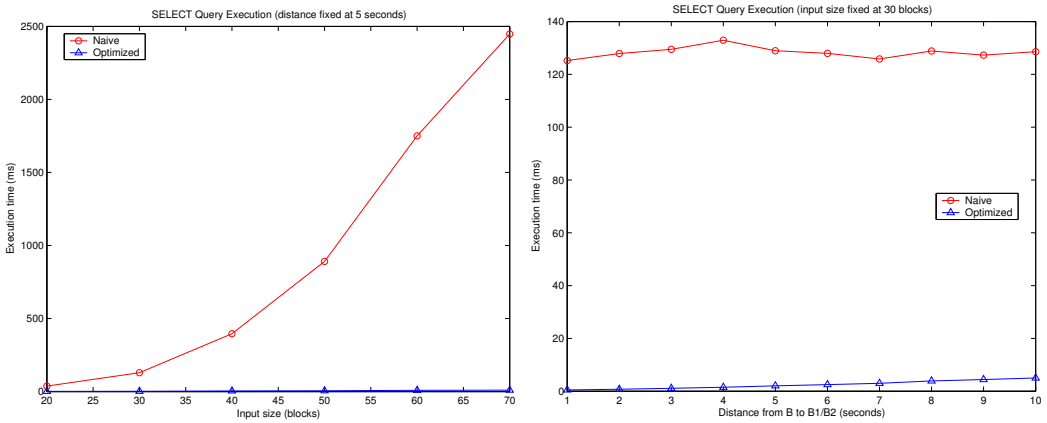


Figure 4.2: Optimized SELECT Performance.

The first experiment compared *selection* queries running with and without optimizations described in Section 4.4. The test program generated SELECT queries of the form

$$\sigma_{in(B_1,name(F,n_1))\wedge in(B_2,name(F,n_2))\wedge before(B,B_1,d)\wedge after(B,B_2,d)}(v),$$

where n_1, n_2 were randomly chosen feature names and the distance d changed from 1 to 10. I first executed 10 such queries with $d = 5$ for input video fragments ranging from 20 to 70 blocks in size. The average execution time of these queries is shown at the left side of Figure 4.2 as a function of the input size. As seen from the graph, the execution time of the original query rises sharply as input size grows, while the optimized query takes almost negligible time in comparison. I then fixed the input size at 30 blocks and varied d from 1 to 10. The right side of Figure 4.2 shows the result of this experiment, where the execution time of the original query does not depend on

d, while the optimized query takes slightly more time as distance constraints become more lax.

The second experiment compared *matching* queries running with and without optimizations, as well as using the R-tree index. The test program picked random sequences of 1 to 5 blocks from the input video, composed a random localized pattern of 1 to 5 features from each selected block, and matched the resulting mini-summary against the input video with k ranging from 1 to 5.

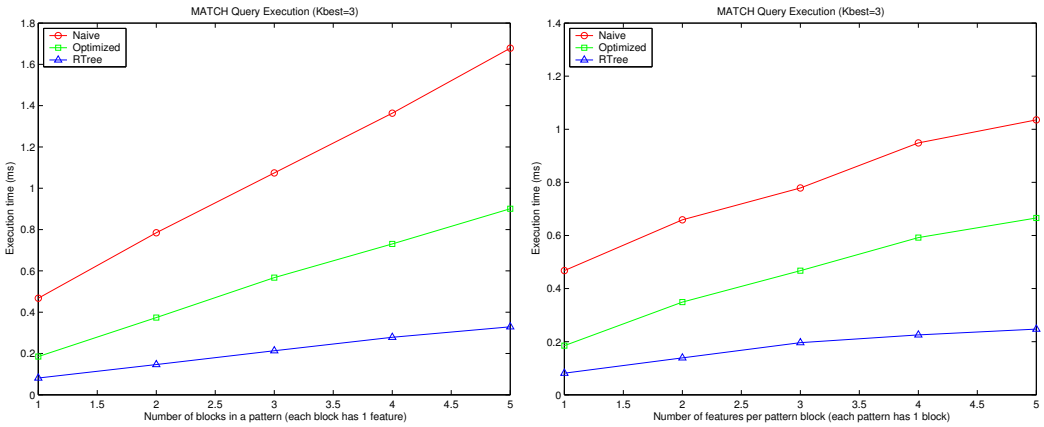


Figure 4.3: Optimized MATCH Performance.

Figure 4.3 shows MATCH execution time as a function of the pattern length and the number of features in each pattern block (it has been found that changes in k do not affect results, as long as k is small). As seen from the figure, the optimized $Match^{+}()$ taking advantage of $bbox$ and $names$ fields performs roughly twice better than the "naive" $Match()$.

I then ran a greedy algorithm that broke each input video block into clusters of closely overlapping features and inserted these clusters into an augmented R-tree. The tree insertion algorithm chooses a node with the the closest fitting $names$ field and then chooses an insertion point with the closest fitting rectangle underneath that

node. This is just one of many possible insertion heuristics though.

The result of running an MATCH query using the R-tree, shown in Figure 4.3, indicates that the query performance can be doubled with respect to $Match^{+}()$ thus making $TreeMatch()$ about four times faster than the original naive $Match()$.

4.5 The Cost Model

After implementing the basic VDA operators, I came up with the cost model, shown in Table 4.1, that recursively predicts the running time of each algebraic operator ($\mathsf{C}()$), the number of blocks in its result ($\mathsf{I}()$), and the average number of featurss in each block ($\mathsf{D}()$).

All formulas in Table 4.1 correspond to queries with $p_{min} = \epsilon$. The JOIN formulas correspond to the *selective* JOIN operator. The set operators used the similarity function $b_1 \sim b_2 \leftrightarrow b_1.t_s = b_2.t_s \wedge b_1.t_e = b_2.t_e$.

The $vars(C)$ function returns the number of variables in C, while the $sel(C)$ function estimates C's selectivity, as percentage of input blocks b (features f) for which $C[B/b]$ ($C[F/f]$) evaluates to a non-zero value. The $cost_{bc}(C, v)$ and $cost_{fc}(C, v)$ functions estimate costs of evaluating a block or feature condition C on a video v. Finally, the value of $cost(tr)$ reflects the cost of applying the transformation function tr.

The k_* constants in Table 4.1 correspond to execution times of various implementation parts and depend on the computer hardware and the operating system. For example, k_{s0} is the cost of copying a single feature from the input of a SELECT operator to its output, and so forth.

The n_* constants depend on the input data. For example, n_{i0} is the percentage of

v_1 that is typically retained by the $v_1 \cap_{\sim} v_2$ operator.

The VDA implementation uses self-tuning to determine both k_* and n_* constants at runtime. Some typical examples of their values are given in the Table 4.2.

To verify the cost model, I have run nine different queries on a hundred of artificially generated video summaries of the same size and computed their average execution times and cost estimates. I then varied input size from 100 to 2000 blocks, each containing $10 - 25$ features and plotted both execution times and cost estimates, as shown in Figure 4.4.

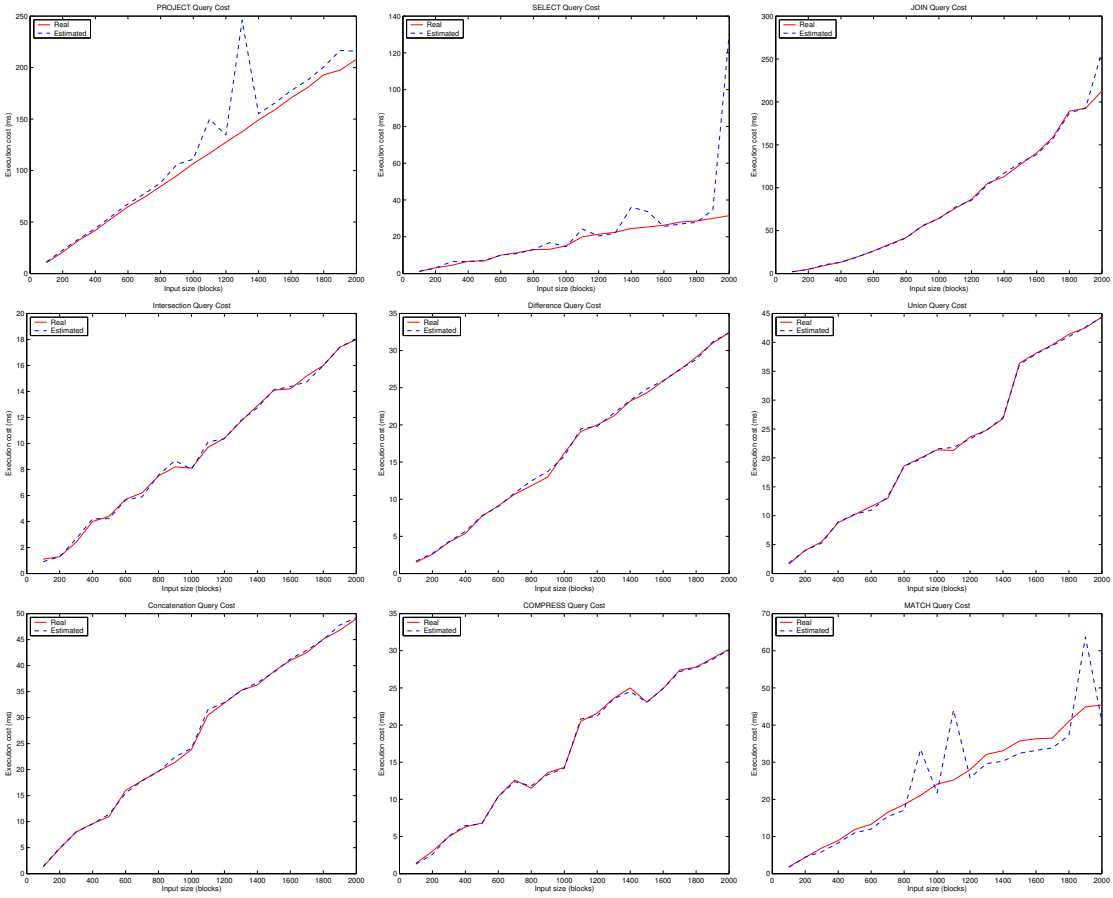


Figure 4.4: Cost Model Verification.

Based on the cost model and equivalences shown in the previous sections of this

chapter, I have implemented a simple query optimizer that uses greedy breadth-first search to rewrite a query in the most optimal way. To test the optimizer, I have run a group of 10 queries on artifically generated video summaries varying from 100 to 2000 blocks in size, each block containing $20 - 50$ features and measured the total running time for the group. The results of this experiment, shown in Figure 4.5, indicate $25 - 30\%$ improvement in the performance of optimized queries.

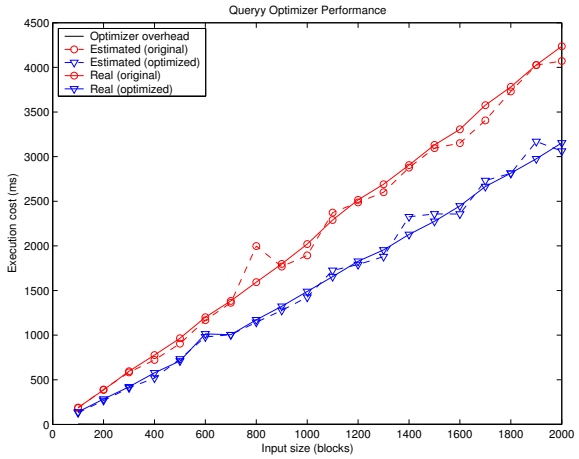


Figure 4.5: Query Optimizer Performance.

4.6 Implementation

I have implemented the VDA Reference System whose architecture is shown in Figure 4.6. The system consists of three main parts: (i) the *feature extraction* component processes video, segments it into blocks, and extracts some elementary features; (ii) the *algebraic core* uses block-feature annotations (both obtained as result of the feature extraction and made by hand) to execute queries; (iii) the *end user interfaces* (CLI and GUI) accept queries from users and present them with the execution results. Let us look at these components in a greater detail.

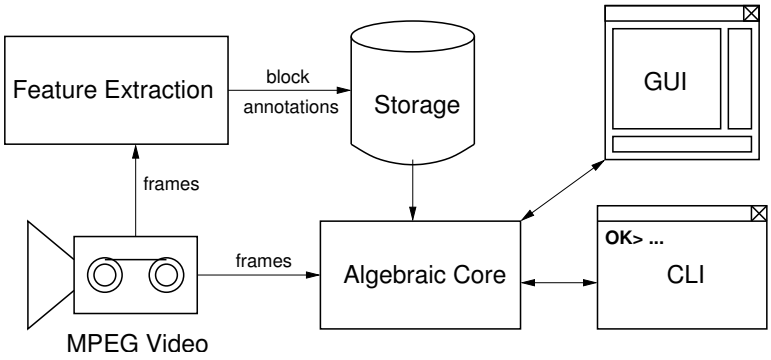


Figure 4.6: VDA Architecture.

4.6.1 Feature Extraction

The feature extraction component of the VDA system can detect two types of features: (i) areas of uniform color and (ii) motion of these areas. It will also break incoming MPEG1 video into blocks based on changes in detected feature combinations and produce a list of blocks with corresponding features. This list, augmented with human-annotated features, can later be used by the VDA algebraic core to query video.

To describe the feature extraction algorithm, I will use following notation:

- b_{next} and b_{prev} are single-frame blocks corresponding to the current and previous video frames.
- b_{acc} is a multi-frame block that we are currently accumulating. A new b_{acc} is started every time there is a significant change in the frame layout, signifying a scene change.
- b_{start} is a single-frame block corresponding to the first frame of b_{acc}.

The feature extraction proceeds frame by frame in following steps:

1. An MPEG1 frame is loaded into memory in `24bpp` format, using the popular MPEGLib decoder.

2. A `15bpp` color histogram of 32768 entries is built and the peaks in this histogram are grown by replacing all colors in their vicinity with the peak colors.

3. The algorithm starts searching the frame for a blob of color c_{max} corresponding to the highest peak in the histogram. When such a blob is found, the algorithm creates an `RGB` feature corresponding to this blob and removes all c_{max}-colored pixels within the blob. If the algorithm fails to find a blob of color c_{max}, it removes *all* c_{max}-colored pixels from the frame. The algorithm then updates the histogram to reflect removed pixels count and proceeds looking for a color corresponding to the next highest peak in the histogram.

4. The search process terminates when most or all pixels have been classified into features and removed from the frame. The result is a frame-long block b_{next} containing a collection of `RGB` features.

5. The algorithm compares b_{next} to a block b_{start} corresponding to the starting frame of b_{acc}.

6. If b_{next} is sufficiently similar to b_{start} in terms of features, it is merged into b_{acc}. During this process, some features in b_{acc} may grow in area, while their coverages decrease.

7. If b_{next} strongly differs from b_{start}, the algorithm decides that it is time to begin accumulating a new block. But first, the b_{acc} contents have to be post-processed and output as result. This is done by converting all `RGB` features in b_{acc} to corresponding color features (such as `RED`, `WHITE`, or `BROWN`) and using the

differences between b_{start} and b_{prev} blocks to detect and add `MOTION` features. The algorithm then prints the contents of b_{acc} to the standard output and resets both b_{acc} and b_{start} to b_{next}.

8. b_{next} is assigned to b_{prev} and the process is repeated until the algorithm runs out of frames.

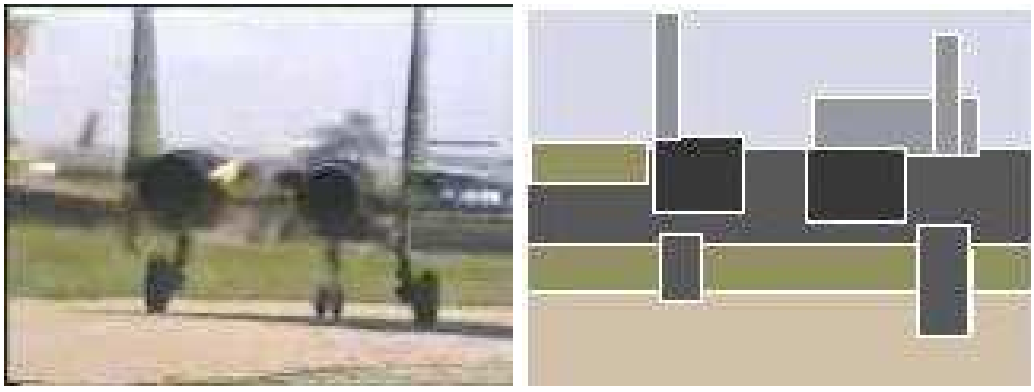


Figure 4.7: Feature Extraction Example.

While the feature extraction algorithm used in the reference VDA implementation is far from perfect, it works quite well for videos that have relatively simple composition (such as cartoons, sports events, or military surveillance), as shown in Figure 4.7. The VDA algebra with also work with any feature extraction system, be it a complicated image segmentation algorithm or a group of humans annotating videos by hand, as long as it produces feature descriptions in the correct format.

4.6.2 Algebraic Core

The VDA algebraic core is implemented as a library of C++ classes representing features, blocks, video summaries, selection conditions, and queries.

Summaries are represented with objects of class `Video` that may contain one or more `Block` objects, ordered by their starting and ending timestamps. Blocks

inside a `Video` object do not have to be contigious, but they cannot overlap thus satisfying the requirement of a video summary definition. `Video` member functions allow programmer to add blocks to a summary, enumerate blocks, and query summary length and block count.

Each `Block` object may contain one or more `Feature` objects, characterized by names, values, coverages, and spatial locations within a frame. `Block` member functions allow programmer to add features to a block, enumerate features, and query block's starting and ending timestamps, length, and feature count.

The two kinds of VDA selection conditions, *feature* and *block* conditions, are represented with `FWFF` and `BWFF` classes accordingly. Each class allows to create condition nodes of all possible types and combine these nodes into trees with logical junctions. Given a set of feature or block variable assignments V, a condition can be evaluated with the $Eval(V)$ method. As many algebraic operators require evaluating conditions over multiple variable assignments, there are also methods $FWFF :: Eval(Feature, Block)$ and $BWFF :: Eval(Block, Video)$ that assign the their first argument to the first variable and go over all possible assignments based on the second argument. Both functions return the maximal $Eval(V)$ value found for all possible assignment sets V.

The virtual class `Query` serves as a base for all classes representing atomic queries. Among other things, the `Query` class defines the method $Create()$ for the query execution, returning a `Video` object with the query result. It is important to note that the `Query` class is derived from `Video`. When addressed as a `Video`, a `Query` object gives access to its last execution result.

A family of classes derived from `Query`, such as `QSelect`, `QProject`, `QJoin`, and so forth, represents atomic queries and allows programmer to build query trees.

The leaves of such a tree are always objects of class `QData` that supply input data from disk, network, or other software, such as image recognition and object tracking systems.

4.6.3 User Interfaces

The VDA system currently has two kinds of user interfaces: the *command line interface* (CLI) and the *graphical user interface* (GUI).

The CLI has been mainly implemented to experiment with the VDA algebraic core and assert its performance, It is a simple tool that uses command line to accept a query, parses and executes this query. Both the query result (a list of blocks) and the execution time are then shown to the user.

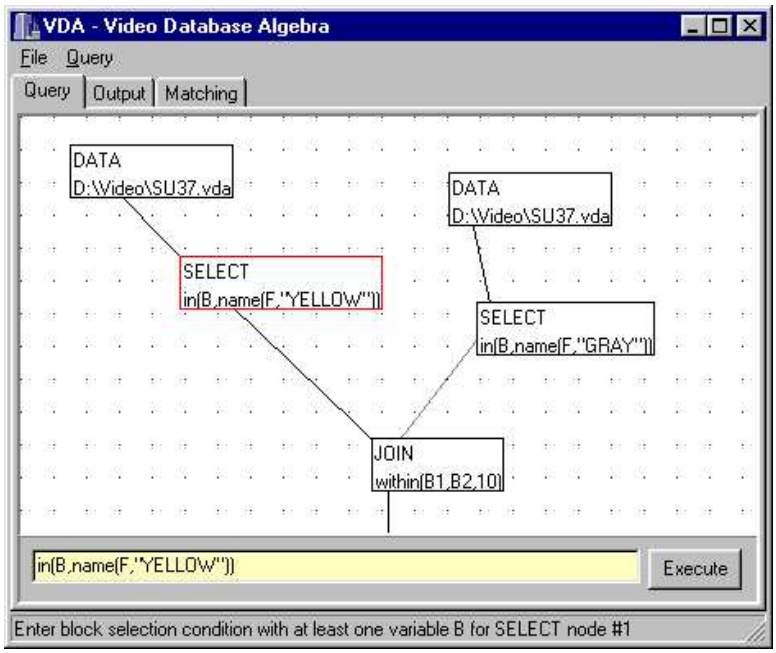


Figure 4.8: Query Composition GUI.

The GUI, shown in Figures 4.8 and 4.9, is a more sophisticated query tool. It

allows users to design their queries graphically and save these queries for later use, as shown in Figure 4.8.

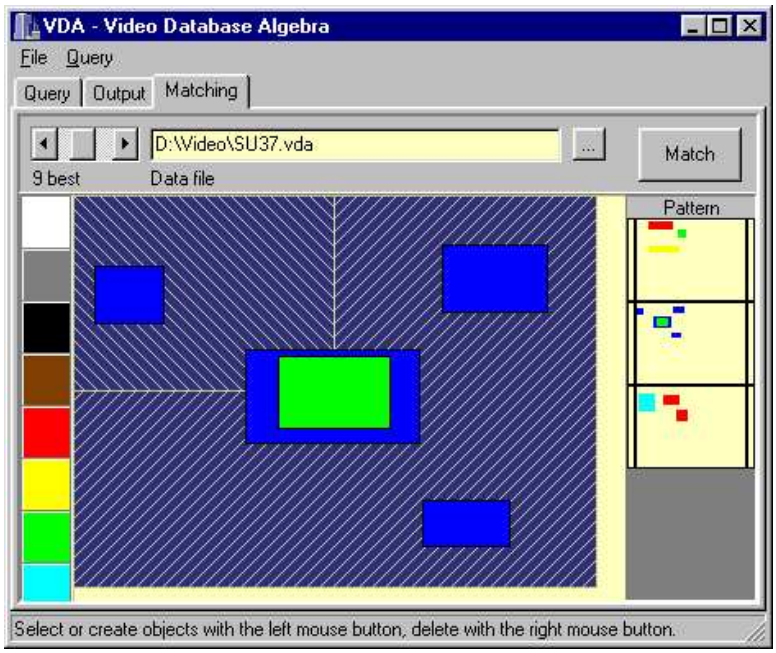


Figure 4.9: Pattern Matching GUI.

In addition, the GUI specifically addresses *pattern matching* tasks by allowing users to sketch feature patterns they are looking for and find them in the input data, as shown in Figure 4.9. Query results are shown to users with a set of DVD-like controls allowing users to play resulting summaries, rewind and fast-forward through blocks, or jump to an arbitrary block.

4.7 Related Work

There exist relatively few works in video algebras. The most prominent ones are [17, 19, 20, 59, 64].

In the OVID project [59], E. Oomoto and K. Tanaka came up with the concept of

a *video object* as a set of attributes defined for an interval of frames. They also introduced an is-a relation over attributes to maintain a semantic hierarchy of attributes. To query videos, the authors proposed a language called VideoSQL. Unfortunately, the proposed algebraic operators, such as *interval projection*, *interval merge*, *interval overlap*, and *object overlap* cannot be easily equated to the traditional relational operators.

The video database algebra by A. Picariello, M.L. Sapino, and V.S. Subrahmanian [64] builds upon the basic notions described in [59] to propose a formal relational-like video algebra. Similarly to OVID, this algebra operates on *objects* characterized by attributes. The attributes *may* (but do not have to) include the *spatial location* of the object and the *time interval* during which the object appears in a video.

While not directly dealing with the videos, an interval-based model for the temporally organized data has been proposed by T.D.C. Little and A. Ghafoor [52]. Later, Y.F. Day, A. Ghafoor et. al. propose an object-based model for video [17] and develop an algebra specifically dealing with videos [19]. This later algebra describes videos in terms of spatial and temporal *events* occurring among *objects* and provides primitives that specify relations between events.

The *Algebraic Video* by A. Duda, R. Weiss, and D.K. Gifford [20] is based upon annotated *video segments* as opposed to objects in [59, 64, 17] or events in [19]. The *hierarchical model* of a video allows segments to overlap in time as long as they belong to different hierarchical branches. Each branch corresponds to a different semantic interpretation of the video with parent nodes corresponding to more abstract interpretations. The proposed algebra provides a rich set of operations to create, annotate, combine, query, and display video segments. Unfortunately, the very richness of the algebraic operators in this algebra makes it difficult to reason about them thus

limiting algebra's theoretical importance.

All video algebras described above either operate on *objects* occuring in frames as opposed to actual frames or segments of a video [59, 64, 17, 19] or use a hierarchical segment structure that complicates reasoning about algebraic operators [20]. The VDA algebra allows queries based on both segments and objects occurring in these segments, yet it is sufficiently simple to preason about, as has been shown in this chapter.

4.8 Conclusions

In this chapter, I have presented an algebra that operates on both video summaries and complete videos. I have formally defined video summaries and basic algebraic operators, shown some useful algebraic equivalences, and discussed indices and algorithms that facilitate selection, join, and pattern matching operators. Furthermore, I have developed and described in this chapter a reference implementation for the VDA algebra, complete with the feature extraction tool, the cost model, the query optimizer, and both command line and graphical user interfaces.

$$
\begin{aligned}
cost_{fc}(C,v) &= k_{fc} \cdot \mathsf{I}(v) \cdot \frac{\mathsf{D}(v)!}{(\mathsf{D}(v) - vars(C))!} \\
cost_{bc}(C,v) &= k_{bc} \cdot \frac{\mathsf{I}(v)!}{(\mathsf{I}(v) - vars(C))!} \\
\mathsf{I}(v) &= card(v) \\
\mathsf{D}(v) &= \frac{\sum_{b \in v} card(b.FS)}{card(v)} \\
\mathsf{C}(v) &= k_{v0} \cdot \sum_{b \in v} card(b.FS) \\
\mathsf{I}(\sigma_C(v)) &= sel(C) \cdot \mathsf{I}(v) \\
\mathsf{D}(\sigma_C(v)) &= \mathsf{D}(v) \\
\mathsf{C}(\sigma_C(v)) &= \mathsf{C}(v) + cost_{bc}(C,v) + k_{s0} \cdot sel(C) \cdot \mathsf{I}(v) \cdot \mathsf{D}(v) \\
\mathsf{I}(\pi_C(v)) &= \mathsf{I}(v) \\
\mathsf{D}(\pi_C(v)) &= sel(C) \cdot \mathsf{D}(v) \\
\mathsf{C}(\pi_C(v)) &= \mathsf{C}(v) + cost_{fc}(C,v) + k_{p0} \cdot sel(C) \cdot \mathsf{I}(v) \cdot \mathsf{D}(v) \\
\mathsf{I}(\beta_C^{tr}(v)) &= \mathsf{I}(v) \\
\mathsf{D}(\beta_C^{tr}(v)) &= \mathsf{D}(v) \\
\mathsf{C}(\beta_C^{tr}(v)) &= \mathsf{C}(v) + cost_{bc}(C,v) + k_{a0} \cdot \mathsf{I}(v) \cdot \mathsf{D}(v) + sel(C) \cdot cost(tr) \cdot \mathsf{I}(v) \\
\mathsf{I}(\alpha_C^{tr}(v)) &= \mathsf{I}(v) \\
\mathsf{D}(\alpha_C^{tr}(v)) &= \mathsf{D}(v) \\
\mathsf{C}(\alpha_C^{tr}(v)) &= \mathsf{C}(v) + cost_{fc}(C,v) + k_{a0} \cdot \mathsf{I}(v) \cdot \mathsf{D}(v) + sel(C) \cdot cost(tr) \cdot \mathsf{I}(v) \cdot \mathsf{D}(v) \\
\mathsf{I}(\mu_k(v_1,v_2)) &= k \cdot \mathsf{I}(v_2) \\
\mathsf{D}(\mu_k(v_1,v_2)) &= \mathsf{D}(v_1) \\
\mathsf{C}(\mu_k(v_1,v_2)) &= \mathsf{C}(v_1) + \mathsf{C}(v_2) + k_{m0} \cdot \mathsf{I}(v_2) \cdot \mathsf{D}(v_1) \cdot \mathsf{D}(v_2) \cdot max(0, \mathsf{I}(v_1) - \mathsf{I}(v_2)) \\
&+ k_{m1} \cdot \mathsf{D}(v_1) \cdot min(\mathsf{I}(v_1), k \cdot \mathsf{I}(v_2)) \\
\mathsf{I}(v_1 \bowtie_C v_2) &= sel(C) \cdot \mathsf{I}(v_1) \\
\mathsf{D}(v_1 \bowtie_C v_2) &= \mathsf{D}(v_1) \\
\mathsf{C}(v_1 \bowtie_C v_2) &= \mathsf{C}(v_1) + \mathsf{C}(v_2) + k_{bc} \cdot n_{j0} \cdot \mathsf{I}(v_1) \cdot \mathsf{I}(v_2) + k_{j0} \cdot sel(C) \cdot \mathsf{I}(v_1) \cdot \mathsf{D}(v_1) \\
\mathsf{I}(v_1 \cap_\sim v_2) &= n_{i0} \cdot \mathsf{I}(v_1) \\
\mathsf{D}(v_1 \cap_\sim v_2) &= \mathsf{D}(v_1) \\
\mathsf{C}(v_1 \cap_\sim v_2) &= \mathsf{C}(v_1) + \mathsf{C}(v_2) + k_{i0} \cdot n_{i0} \cdot \mathsf{I}(v_1) \cdot \mathsf{D}(v_1) \\
\mathsf{I}(v_1 \setminus_\sim v_2) &= n_{d0} \cdot \mathsf{I}(v_1) \\
\mathsf{D}(v_1 \setminus_\sim v_2) &= \mathsf{D}(v_1) \\
\mathsf{C}(v_1 \setminus_\sim v_2) &= \mathsf{C}(v_1) + \mathsf{C}(v_2) + k_{d0} \cdot n_{d0} \cdot \mathsf{I}(v_1) \cdot \mathsf{D}(v_1)
\end{aligned}
$$

$k_{v0} = 0.000049$	$k_{s0} = 0.000379$	$k_{p0} = 0.0687801$	$k_{m0} = 0.000011$
$k_{m1} = 8.57e-09$	$k_{j0} = 0.000782$	$k_{i0} = 0.0000067$	$k_{d0} = 0.000302$
$k_{u0} = 0.000337$	$k_{c0} = 0.000343$	$k_{r0} = 0.0004899$	$k_{a0} = 0.000574$
$k_{r0} = 0.000439$	$k_{c0} = 0.000357$	$k_{m0} = 0.0000178$	$k_{s0} = 0.0045847$
$n_{i0} = 0.028451$	$n_{d0} = 0.971549$	$n_{u0} = 0.459769$	

Table 4.2: VDA Cost Model Constants.

Chapter 5

Summarizing Video

5.1 Introduction

When querying a video, or a database of videos, it seems natural that the user would like to receive results of his query as one or more short clips, or *summaries*, containing only the information requested in the query. For example, a TV sports commentator reporting on a soccer game may wish to see all the goals in that game. A student studying from home would ask for the part of a lecture video where the professor is talking about a certain topic or assigning homework. A security guard checking on security videos would request all the parts where movement occured in a doorway. In all these cases, users would receive a new video, much shorter than the original one, that is limited to the requested information. In some cases, users may have more complicated requirements for video summaries. For example, the sports commentator may not want to see goal replays but *would* like to see what preceded each goal. The security guard might need to omit parts of the tape where he himself is on camera.

The work presented in this chapter [4, 24] is based on the *continuity-priority-repetition* (CPR) model that rates video summaries, with respect to three characteris-

tics as follows:

1. *Continuity:* A summary with a lot of "jumps" between shots is unlikely to be attractive to users. Thus, it must be as continuous as possible.

2. *Priority:* For each given application, certain objects or events shown in the video may be more important than others. For example, when summarizing soccer videos, a goal is more important than a midfield pass. A summary composition must favor high priority features, as defined by the application.

3. *Repetition:* Even though a feature may have high priority, it may be undesirable to repeat it over and over again, while ignoring other features. Thus, a summary has to be non-repetitive.

These three considerations form the core basis for the proposed summarization framework.

The CPR model [24] consists of two key components: (i) use of rules to specify which features in a video are of interest (i.e. have high priority) for inclusion in a summary and (ii) an objective function that balances the relative importance of the content with its continuity and repetition. Once the rules and the objective function are articulated, *any* suite of video processing algorithms can be used for feature extraction.

This chapter starts by introducing concepts of features, frames, blocks, videos, and summaries, and proceeds to formulate a way to specify, with a set of rules, the desired summary content. To evaluate summary worth, we define the CPR-based *evaluation function* and show several ways in which its components can be computed. Based on this evaluation function, the problem of creating "optimal" summaries is introduced.

Next, I go over algorithms to create optimal summaries proposed by the group of A. Picariello at the University of Naples and then introduce my own approach, known as the Summary Extension Algorithm (SEA). I present a variety of increasingly complex versions of the SEA algorithm, designed to improve its performance. The last algorithm presented in this chapter is the Priority Curve Algorithm (PCA). This algorithm uses implicit assumptions about CPR criteria instead of relying on the evaluation function.

Following discussion of the algorithms, I present experimental results obtained at the University of Naples. About 200 students there have tested our algorithms on 50 soccer videos with the goal of determining which algorithm produced the best subjective summary quality. It has been concluded that the PCA algorithm produces the best summaries in the shortest time, followed by the SEA algorithm.

The chapter concludes with the discussion of prior works in the video summarization field and the differences between these works and our work.

5.2 Formal Model of Video Summarization

A *video* v is a sequence of frames. In many cases, one may want to coalesce groups of contiguous frames into *blocks* and then create summaries based on blocks rather than frames. The advantage of this approach is that the number of blocks in a video is much smaller than the number of frames. Independently of which approach is used, the framework described below applies both to frames and blocks. For the sake of generality, I will proceed using blocks, while pointing out the difference between two approaches wherever it exists.

Each frame or block can be characterized by a variety of properties, such as move-

ment in a certain part of the frame, a certain colors, or the presence of certain objects, events, or actions. Let us call such properties *features* and define the above concepts formally:

Definition 5.2.1 (Block) *A video* block *is a structure* $b = \langle t_s, t_e, F \rangle$ *where* $b.t_s$ *is the starting frame number,* $b.t_e > b.t_s$ *is the ending frame number, and* $b.F$ *is a set of features occuring in* b*.*

The *length* of a block $length(b) = b.t_e - b.t_s$.

Definition 5.2.2 (Video) *A* video *is a finite set of blocks* $v = \{b_1, \ldots, b_n\}$ *such that* $\forall 1 \leq i < n : \ b_{i+1}.t_s = b_i.t_e$*.*

Given the above definition of a video, we can define a total ordering on blocks in a video as $\forall 1 \leq i, j \leq n : \ b_i \leq b_j \leftrightarrow b_i.t_e \leq b_j.t_s$. The *length* of a video $length(v) = b_n.t_e - b_1.t_s$, while the number of blocks in a video is simply $card(v)$. When all blocks have a uniform length, $length(v)$ becomes proportional to $card(v)$ and thus they can be easily interchanged. Same happens when frames are used instead of blocks, as we can treat them as blocks of uniform length 1.

Let us now define a video *summary* as a subset of a video:

Definition 5.2.3 (Summary) *A* summary *of a video* v *is a set of blocks* $s = \{b_1, \ldots, b_n\}$ *such that* $s \subseteq v$ *and* $\forall 1 \leq i < n : \ b_{i+1}.t_s \geq b_i.t_e$*.*

Given above definitions of videos and summaries, one can easily show that any video is also a summary of itself and that the total ordering defined for blocks in videos also applies to blocks in summaries. The *length* of a summary $length(s) = \sum_{b_i \in s} length(b_i)$.

5.3 Video Summaries

For the user working with the video, a typical task would be to create a video summary whose blocks contain certain features. For example, the policeman processing an ATM security video may want to see blocks that contain more than two people at the same time. The sports commentator may create a summary of a soccer match that shows all the goals in this match, and so forth.

Below, I will try to come up with the framework allowing to query videos, rate query results, and create an optimal summary by user's request. The rest of this chapter assumes that there is some video v that we want to summarize and all the video blocks are coming from this video. Let us further assume that all video is stored in a video database supporting following access methods:

- findblocks(v, f): Given a video v and a feature f, this function returns the set of all blocks in v that contain f.
- findfeatures(v, b): Given a video v and a block $b \in v$, this function returns the set of all features in b.

Most existing video databases, such as AVIS[2] and OVID[59], can support such functions. Note that all the above functions return *sets* as output.

Let us start on the summary content specification language by defining *block-coverage pairs*.

Definition 5.3.1 (Block Coverage Pair) *Let b be a video block and $p \in [0, 1]$ be a real value. Then $\langle b, p \rangle$ is a* block-coverage pair *or* BCP .

I will also call p the *coverage*. As shown later, the coverage represents how well b satisfies conditions imposed on it. A summary can be represented by a set of block-coverage pairs. I will use the upper case S for this representation of a summary, as

opposed to the lower case s for a set of blocks. A union of two BCP sets is somewhat different from the canonical set union.

Definition 5.3.2 (BCP Set Union) *Given two sets of block-coverage pairs V_1, V_2, the* BCP set union

$$V_1 \cup V_2 = \{\langle b, p\rangle \mid p = \begin{cases} p_1 & if\ \langle b, p_1\rangle \in V_1 \wedge\ \nexists \langle b, p_2\rangle \in V_2 \\ p_2 & if\ \langle b, p_2\rangle \in V_2 \wedge\ \nexists \langle b, p_1\rangle \in V_1 \\ max(p_1, p_2) & if\ \langle b, p_1\rangle \in V_1 \wedge \langle b, p_2\rangle \in V_2 \end{cases}\}.$$

For further definitions, let us assume the existance of a set V_b, of all variables ranging over BCP pairs and a set V_f of all variables ranging over features. Members of V_b are known as *block variables* while members of V_f are known as *feature variables*. Let us now define the *video calls* and the *atoms*.

Definition 5.3.3 (Video Call) *Suppose vc is a video database API function, and $t_1, \ldots, t_n$ are either arguments to vc (of the right type) or variables ranging over the values of the appropriate type. Then $vc(t_1, \ldots, t_n)$ is called a* video call.

Definition 5.3.4 (Atoms)

1. *Given a block constant or variable X,* $\mathsf{insum}(X)$ *is a* membership atom.
2. *Given a video call $vc(t_1, \ldots, t_n)$ and a constant or a variable X of the same type as members of vc's output set, $X \in vc(t_1, \ldots, t_n)$ is a* feature atom.
3. *Given two block constants or variables X, Y and an integer value $n \in [0, +\infty)$,* $\mathsf{near}(X, Y, n)$, $\mathsf{before}(X, Y, n)$, *and* $\mathsf{after}(X, Y, n)$ *are* sequence atoms.

Intuitively, membership atoms are used to require the presence of certain blocks in a summary, or bind a variable to summary blocks. For instance, $\mathsf{insum}(X)$ binds

variable X to all blocks in a summary. Feature atoms are used to require the presence of a certain feature in a block, or to bind a variable to features in a block. For example, $X \in \mathsf{findblocks}(v, \text{“}Human\text{”})$ binds X to any block that contains humans. Similarly, $\text{“}Red\text{”} \in \mathsf{findfeatures}(v, Y)$ requires red color to be present in a block Y. Sequence atoms are used to ensure continuity by requiring that some blocks occur near each other and in a certain order. Intuitively, a block X satisfies $\mathsf{before}(X, Y, d)$ iff X ends in the the interval of frames starting at $Y.t_s - d$ and ending at $Y.t_s$. $\mathsf{after}(X, Y, d)$ is equivalent to $\mathsf{before}(Y, X, d)$ and $\mathsf{near}(X, Y, d)$ is equivalent to $\mathsf{after}(X, Y, d) \vee \mathsf{before}(X, Y, d)$.

Definition 5.3.5 (Variable Assignment Set) *If X is a block variable and $\langle b, p\rangle$ is a* BCP *then $X/\langle b, p\rangle$ is an* assignment. *The* assignment set *θ is a set of assignments such that*

$$\forall X/\langle b, p\rangle \in \theta : \ \forall Y/\langle b', p'\rangle \in \theta : \ X = Y \rightarrow b = b' \wedge p = p'.$$

Given an atom a and an assignment set θ, I will use $a\theta$ to express the *substitution* of variables in a by their values in θ.

Definition 5.3.6 (Interpretation) *Given a set D of all possible block-coverage pairs, a summary $S \subset D$, a feature f, a variable assignment set $\theta = \{X/x, Y/y\}$, and an atom $a \in \{\mathsf{insum}(X), X \in \mathsf{findblocks}(v, f), f \in \mathsf{findfeatures}(v, X), \mathsf{after}(X, Y, n), \mathsf{before}(X, Y, n), \mathsf{near}(X, Y, n)\}$, let us define the real-valued function $\lambda(a\theta, S) \rightarrow [0, 1]$ such that*

$$\lambda(\mathsf{insum}(x), S) = \begin{cases} x.p & if \ \exists \langle b, p\rangle \in S : \ x.b = b \wedge x.p = p \\ 0 & otherwise \end{cases}$$

$$\lambda(x \in \mathsf{findblocks}(v, f)) = \begin{cases} x.p & if \ x.b \in \mathsf{findblocks}(v, f) \\ 0 & otherwise \end{cases}$$

$$\lambda(f \in \mathsf{findfeatures}(v,x)) = \begin{cases} x.p & if\ f \in \mathsf{findfeatures}(v, x.b) \\ 0 & otherwise \end{cases}$$

$$\lambda(\mathsf{after}(x,y,n),S) = min(x.p, y.p) \cdot \begin{cases} \frac{n - x.b.t_s + y.b.t_e}{n} & if\ x.b.t_s - y.b.t_e \in [0,n] \\ 0 & otherwise \end{cases}$$

$$\lambda(\mathsf{before}(x,y,n),S) = \lambda(\mathsf{after}(y,x,n),S)$$

$$\lambda(\mathsf{near}(x,y,n),S) = \max(\lambda(\mathsf{before}(x,y,n),S), \lambda(\mathsf{after}(x,y,n),S))$$

It is also possible to make API functions themselves return coverages and use them to interpret feature atoms. To simplify things though, I will assume that API functions return sets of plain blocks or features.

Example 5.3.1 (Feature Atom Interpetation) *Consider block* $b = \langle 4, 5, \{f_1, f_2, f_3\}\rangle$. $\lambda(f_1 \in \mathsf{findfeatures}(v, \langle b, 0.5\rangle))$ *evaluates to* 0.5*, as* b *contains* f_1*. On the other hand, if we replace* f_1 *with some feature* $f_4 \notin b.F$ *then* $\lambda(f_4 \in \mathsf{findfeatures}(v, \langle b, 0.5\rangle, S)$ *evaluates to* 0*. Notice that the* S *argument is not used to evaluate feature atoms.*

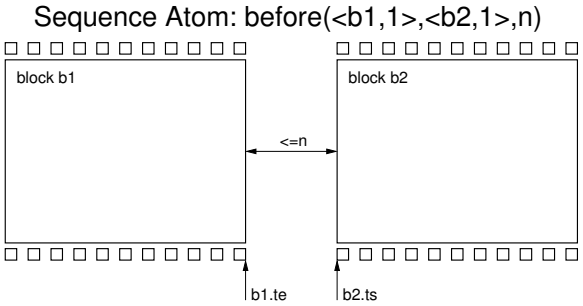


Figure 5.1: Sequence Atom Interpretation.

Example 5.3.2 (Sequence Atom Interpretation) *Consider blocks* $b_1 = \langle 4, 5, F_1\rangle$*,* $b_2 = \langle 7, 9, F_2\rangle$*, and* $b_3 = \langle 10, 13, F_3\rangle$*. Let's evaluate some sequence atoms on these blocks:*

$$\lambda(\mathsf{before}(\langle b_1, 1\rangle, \langle b_2, 1\rangle, 10), S) = 1 \cdot 1 \cdot (10 - 7 + 5)/10 = 0.8$$

$$\lambda(\mathsf{before}(\langle b_1, 1\rangle, \langle b_3, 1\rangle, 10), S) = 1 \cdot 1 \cdot (10 - 10 + 5)/10 = 0.5$$

$$\lambda(\mathsf{after}(\langle b_1, 1\rangle, \langle b_2, 1\rangle, 10), S) = 0 \; as \; b_1.t_s - b_2.t_e = 4 - 9 = -5 < 0$$

$$\lambda(\mathsf{after}(\langle b_2, 1\rangle, \langle b_3, 1\rangle, 10), S) = 0 \; as \; b_2.t_s - b_3.t_e = 7 - 13 = -6 < 0$$

Notice that the S argument is not used to evaluate sequence atoms. By looking at the first two evaluations (also shown at the Figure 5.1) one can see that the value of p decreases as the distance between blocks grows, until p becomes 0 for distances $\geq n$. The last two evaluations show that the order of blocks is also important.

We can now define *rules* and *rule sets*:

Definition 5.3.7 (Rule) *A* block selection rule r *is an expression of this form:*

$$\underbrace{[w]}_{weight(r)} \quad \underbrace{insum(X)}_{head(r)} \leftarrow \underbrace{a_1 \wedge \ldots \wedge a_n}_{body(r)}$$

where $w \in [0, 1]$ is a real value, X is a block variable, and $a_1, \ldots, a_n$ are atoms such that for each block variable $Y \in r$ there is a membership atom $insum(Y) \in r$.

Example 5.3.3 (Rule) *The following is an example of a rule:*

$$[0.9]\ \mathsf{insum}(Y) \leftarrow X \in \mathsf{findblocks}(v, \text{“}Red\text{”}) \wedge Y \in \mathsf{findblocks}(v, \text{“}Human\text{”}) \wedge \mathsf{near}(X, Y, 10) \wedge \mathsf{insum}(X).$$

This is not *a rule as it misses the* $\mathsf{insum}(Z)$ *atom in the body:*

$$[1]\ \mathsf{insum}(X) \leftarrow \text{“}Green\text{”} \in \mathsf{findfeatures}(v, X) \wedge \text{“}Human\text{”} \in \mathsf{findfeatures}(v, Z) \wedge \mathsf{after}(Z, X, 15).$$

Definition 5.3.8 (Rule Satisfaction) *Suppose r is a rule, S is a summary, X is a block variable occuring in $head(r)$, b is a video block, and θ is a variable assignment set. Let us define a real-valued function $\lambda(r, b, S) \rightarrow [0, 1]$ such that*

$$\lambda(r, b, S) = weight(r) \cdot \max\{\min\{\lambda(a_i\theta, S) | a_i \in body(r)\} \mid \theta \; is \; a \; variable \; assignment\}$$

The block-coverage pair $\langle b, \lambda(r, b, S)\rangle$ is said to satisfy *r w.r.t. S iff $\lambda(r, b, S) > 0$.*

Intuitively, the satisfaction of a rule by $\langle b, \lambda(r, b, S)\rangle$ means that b can be included into the summary S. $\lambda(r, b, S)$ measures how desirable b is w.r.t. to the rule and the other blocks in the summary. $weight(r)$ measures the importance of the rule relative to other rules.

Definition 5.3.9 (Rule Set) *A* rule set *is a set of block selection rules.*

Definition 5.3.10 (Rule Set Satisfaction) *Suppose R is a rule set, b is a video block, and S is a summary. Let us define a real-valued function $\lambda(R, b, S) \rightarrow [0, 1]$ such that*

$$\lambda(R, b, S) = \max_{r \in R} \lambda(r, b, S).$$

The block-coverage pair $\langle b, \lambda(R, b, S)\rangle$ is said to satisfy *R w.r.t. S iff $\lambda(R, b, S) > 0$.*

Intuitively, any block that satisfies the rule set with respect to some other blocks in the summary can be considered for inclusion into the summary. Hence is the definition of a *satisfactory summary*:

Definition 5.3.11 (Satisfactory Summary) *Given a rule set R, a summary S is called a* satisfactory *w.r.t. R iff*

$$\forall \langle b, p\rangle \in S : \ \langle b, p\rangle \ satisfies \ R \ w.r.t. \ S$$

One can also say that the satisfactory summary is a set of all blocks occuring in S.

Notice that the satisfactory summary definition does not specify what the summary length should be. In fact, the whole video may well be a satisfactory summary of itself. Let us now put restrictions on the summary length:

Definition 5.3.12 (k-Summary and l-Summary) *Given an integer value $k \in [0, +\infty)$, a summary S is called a k-*summary *iff $card(S) \leq k$. Given a real value $l \in [0, +\infty)$, a summary S is called an l-*summary *iff $length(S) \leq l$.*

Notice that when using frames or uniform-length blocks to represent a video, l-summary and k-summary effectively become the same thing. This work focuses on computing k-summaries. It is clear that the above definition allows for multiple satisfactory k-summaries. Some of them may be better than others with respect to user's needs. Let us then define the *optimal k-summary*:

Definition 5.3.13 (Optimal k-Summary) *Given a set of all satisfactory k-summaries $\mathcal{S}$ and a function $eval : \mathcal{S} \rightarrow [0, +\infty)$, the* optimal k-summary *is a satisfactory k-summary $S \in \mathcal{S}$ such that $\forall S' \in \mathcal{S} : \ eval(S') \leq eval(S)$.*

5.4 Summary Evaluation Function

According to the CPR model, the summary evaluation function $eval()$ used to rate summary quality has to take into account *continuity* ($con()$), *priority* ($pri()$), and *repetition* ($rep()$) of its input. Hence is the following formula:

$$eval(S) = w_c \cdot con(S) + w_p \cdot pri(S) + w_r \cdot (1 - rep(S)).$$

It is assumed that the values of individual components are normalized to the $[0, 1]$ range and w_c, w_p, w_r are weights set according to user's preferences.

The most important consideration when computing a summary is how *appropriate* the summary content is to the user. Given that the p value inside each block-coverage pair expresses block relevance, it is quite easy to come up with a function that measures summary's total *relevance*, as shown in the following example.

Example 5.4.1 (Rule-Based Priority) *This function measures the priority of a summary S based on coverage values of its individual blocks:*

$$pri(S) = \frac{\sum_{\langle b,p \rangle \in S} p}{card(S)}.$$

Of course, there are other ways to compute priority, such as a simple assignment of priorities to all blocks in a video:

Example 5.4.2 (Singular Tabular Priority) *In this method, we have a table with the schema* $(Block, Priority)$*. An example of such a table may contain tuples* $\langle b_1, 5\rangle$*,* $\langle b_2, 3\rangle$*, and* $\langle b_3, 4\rangle$*. Given a block* b*, the priority* $pri(b)$ *of the block is obtained by consulting the table. For a summary* S*, we may define*

$$pri(S) = \sum_{\langle b,p\rangle \in S} pri(b).$$

Thus, for instance, $pri(\{\langle b_1, p_1\rangle, \langle b_2, p_2\rangle\}) = 5 + 3 = 8$*.*

One can also assign priorities to the groups of blocks, as follows:

Example 5.4.3 (Aggregated Tabular Priority) *In this method, we have a table with the schema* $(BlockSet, Priority)$*. The first column of this table now contains a* set *of blocks. An example of such a table may contain tuples* $\langle\{b_1, b_2\}, 5\rangle$*,* $\langle\{b_1\}, 3\rangle$*, and* $\langle\{b_2, b_3\}, 7\rangle$*. Given such a table and a summary* S*, many different priority functions may be defined, such as the subset average or the maximal subset average, for example. The* subset average *function finds all tuples in the table whose* $BlockSet$ *field is a subset of* S *and returns the average of the priority fields of such tuples. For example, with respect to the above table, if* $B = \{b_1, b_2, b_4\}$*, this function would return* 4 *(average of* 5 *and* 3*). The* maximal subset average *finds all tuples* t *in the table whose* $BlockSet$ *field is a* maximal *subset of* S *(i.e. there is no other tuple* t' *with* $t.BlockSet \subset t'.BlockSet$ *such that* $t'.BlockSet \subseteq S$*) and takes the average priorities of such tuples. In the above example, if* $S = \{\langle b_1, p_1\rangle, \langle b_2, p_2\rangle, \langle b_3, p_3\rangle\}$*, then this priority function would return* 6 *(average of* 5 *and* 7*). Note that the second tuple would not be maximal and hence its associated priority would not be involved in the average computation.*

Computing other two measures is somewhat more difficult as continuity and repetition can be interpreted differently by different people. For example, one may think of *continuity* in terms of the uniform distance between consequent blocks.

Example 5.4.4 (Uniform Block Distance Continuity) *This measure uses the standard deviation of the block distance to express continuity:*

$$\begin{aligned} d_{avg} &= \frac{\sum_{\langle b_i,p_i\rangle,\langle b_{i+1},p_{i+1}\rangle \in S}(b_{i+1}.t_s - b_i.t_e)}{card(S)-1}, \\ con(S) &= 1 - \frac{\sum_{\langle b_i,p_i\rangle,\langle b_{i+1},p_{i+1}\rangle \in S}(b_{i+1}.t_s - b_i.t_e - d_{avg})^2}{d_{avg}^2(card(S)-1)}. \end{aligned}$$

Another way to think about continuity may involve the uniformity of selected block lengths.

Example 5.4.5 (Uniform Block Length Continuity) *This measure uses the standard deviation of the block length to express continuity:*

$$\begin{aligned} l_{avg} &= \frac{\sum_{\langle b_i,p_i\rangle \in S}(b_i.t_e - b_i.t_s)}{card(S)}, \\ con(S) &= 1 - \frac{\sum_{\langle b_i,p_i\rangle \in S}(b_i.t_e - b_i.t_s - l_{avg})^2}{l_{avg}^2 card(S)}. \end{aligned}$$

In addition to analyzing block distances and lengths, one can also use image processing methods to measure continuity, as follows.

Example 5.4.6 (Histogram Distance Continuity) *Suppose $H(f) = (h_1, h_2, \ldots, h_n)$ is a function that returns the color histogram for a given frame number f. Each h_j corresponds to the number of pixels in a region of some color space. A good perceptually uniform space is the* Hue Saturation Value (HSV) *space or alternatively we may use the* Opponent Colors *space. Let d be any measure of distance between two*

histograms (e.g. d could be the well known L_1 or L_2 norms). Now set the continuity of a summary $S = \{\langle b_1, p_1\rangle, \ldots, \langle b_k, p_k\rangle\}$ to be

$$con(S) = \frac{card(S) - 1}{\sum_{i=1}^{card(S)-1} d(H(b_i.t_e - 1), H(b_{i+1}.t_s))}.$$

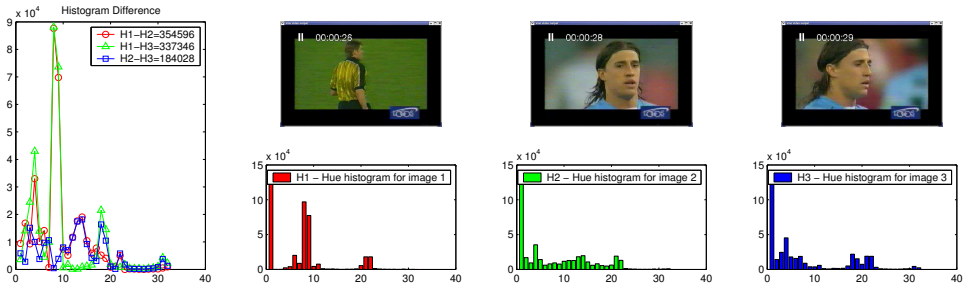


Figure 5.2: Hue Histogram Distance Function.

Figure 5.2 shows three sample frames with respective *hue* histograms and the distance between these histograms. It is clear that the distance between frames 2–3 is less than the distance between frames 1–2 and frames 1–3. Thus, frame 2 preceding frame 3 in a summary will lead to a better continuity.

Finally, the *repetition* can be expressed as the inverse ratio between the number of *different* features in the summary and the *total* number of features. Or, it may depend on the ratio between the number of features in the summary and the entire video, as shown in the following example.

Example 5.4.7 (Feature-Based Repetition) *This metric measures the variety of features included into a summary:*

$$rep(S) = 1 - \frac{card(\{f \mid f \in \bigcup_{\langle b,p\rangle \in S} b.F\})}{card(\{f \mid f \in \bigcup_{b \in v} b.F\})}.$$

5.5 Creating Summaries

The definition of a satisfactory summary allows for multiple satisfactory summaries of k blocks and less. But how do we create such summaries? And how do we choose the best one?

In the discussion of insum-atoms, it was said that $\lambda(\mathsf{insum}(b), S)$ evaluates to a non-zero value whenever block b is in S. When composing a summary, we can assume that (i) S contains blocks that are *candidates* for the final summary or that (ii) S contains blocks that are already *selected* for the final summary. The first approach allows us to obtain a set of all possible summary candidates *before* combining them into summaries, in a following way:

```
Algorithm Der(v,R)
    v is a video
    R is a rule set
begin
    S := ∅
    repeat
        Δ := ∅
        for each block b ∈ v do
            for each rule r ∈ R do
                if λ(r, b, S) > 0 then Δ := Δ ∪ {⟨b, λ(r, b, S)⟩}
            end for
        end for
        S := S ∪ Δ
    until Δ = ∅
    return S
end
```

The Der execution time is quadratic w.r.t. the number of blocks in v and the number of rules in R. The first three algorithms in this section use the first insum interpretation by calling Der once to obtain a set of all candidate blocks. Unfortunately, this may lead to selecting blocks that have no basis to be in the summary. For example, an after-goal celebration may be selected in the absence of a goal that has caused it. This problem can be partially avoided by rating such "inconsistent" summaries down in

the evaluation function, or fully avoided by using the second approach, where insum-atoms are evaluated w.r.t. a set of blocks already selected for the summary (as done in the SEA algorithm).

Let us start by introducing a summarization algorithm called CPRopt which finds an optimal k-summary without making any assumptions about the priority, continuity, and repetition functions. However, as the optimal k-summary computation problem is NP-complete (by reducing knapsack problem to it), this algorithm takes an exponential amount of time w.r.t. the number of blocks in a video, which is clearly unacceptable.

5.5.1 The Optimal Summarization Algorithm

The CPRopt algorithm starts by computing the set of all candidate blocks with $\mathsf{Der}(v, R)$. This step can be executed in time quadratic to the number of frames in v. The algorithm then considers all subsets of $\mathsf{Der}(v, R)$ that contain k or less frames. The $eval()$ function is applied to these subsets and the one with the maximal $eval()$ value is chosen. As the set of all subsets of size $\leq k$ needs to be enumerated, CPRopt has exponential time complexity – this is not a surprise as the problem of finding an optimal summary has been shown to be NP-complete.

```
Algorithm CPRopt(v,R,k)
    v is a video
    R is a rule set
    k is a desired summary length
begin
    V := Der(v, R)
    BestS := ∅
    for each S ∈ {X | X ⊆ V ∧ card(X) ≤ k} do
        if eval(S) > eval(BestS) then BestS := S
    end for
    return BestS
end.
```

Although the CPRopt algorithm is guaranteed to find the optimal summary w.r.t.

the evaluation function, its NP-completeness makes it quite useless for any practical purposes. A. Picariello's group has proposed two heuristic algorithms that are much faster than the CPRopt algorithm, at the cost of producing suboptimal summaries. These are dynamic programming based CPRdyn and genetic programming based CPRgen algorithms.

5.5.2 The CPRdyn Algorithm

The CPRdyn algorithm is based on the *dynamic programming* approach [16]. The algorithm maintains a variable S describing the best solution found so far. Initially, S consists of k randomly chosen blocks which are derivable from the rule set. The algorithm changes S in each iteration by checking to see whether replacing a block in S by a block which is not in S will lead to a better summary. The space complexity of CPRdyn is linear in the number of block, while the time complexity is exponential in k (which is much better than being exponential in the number of blocks).

Algorithm CPRdyn(v,$\mathcal{V}$,k)
  v is a video
  R is a rule set
  k is a desired summary length
begin
  // Fill S with k randomly selected blocks from Der(v, R).
  $V :=$ Der(v, R)
  $S := \{\langle b_i, p_i\rangle \mid i \in [1, k] \ \wedge \ \langle b_i, p_i\rangle \in V\}$
  // Leave the remaining blocks in V.
  $V := V \setminus S$
  while $V \neq \emptyset$
    $subs := false$
    $r := 1$
    while $r < k$ and $subs = false$
      // Build a new tentative solution by replacing $\langle b_r, p_r\rangle$ with a block from V.
      $S' := (S \setminus \{\langle b_r, p_r\rangle\}) \cup \{first(V)\}$
      if $eval(S) < eval(S')$ then
        $S := S'$
        add $\langle b_r, p_r\rangle$ to the tail of V
        $subs := true$
      else
        $r := r + 1$

```
            end if
        end while
        V := V \ {first(V)}
    end while
    return S
end.
```

It is important to note that the CPRdyn algorithm will only consider summaries whose length is exactly k blocks. While this may look like a serious limitation, it is not, as long as the $eval()$ function used in the algorithm is monotonic. Consider a summary S and a block $b \in S$. When $eval()$ is monotonic, it will always be true that $eval(S \setminus \{\langle b, p\rangle\}) \leq eval(S)$.

5.5.3 The CPRgen Algorithm

The CPRgen algorithm uses the *genetic programming* approach [16] to compute a k-summary. The algorithm starts by creating a population of randomly generated summaries and rates population members according to the value of $eval()$. A mutation operator is then applied to a randomly chosen population member, and the member with the smallest $eval()$ value is eliminated. The algorithm stops when the variation of the $eval()$ values within the population falls below a threshold δ.

```
Algorithm CPRGen(v,R,k,N,δ)
    v is a video
    R is a rule set
    k is a desired summary length
    N is the desired number of iterations
    δ is the desired worth threshold
begin
    M := ⌈card(v)/k⌉
    // Compute an initial population of M random solutions
    V := {S_i ⊆ Der(v,R) | i ∈ [1,M] ∧ card(S_i) = k}
    for j ∈ [1,N]
        for i ∈ [1,M]
            S := a solution randomly chosen among the ones in V
            Select a random block b ∈ S
            Choose another block b' ∉ S
            S := (S \ {b}) ∪ {b'}
```

```
            V := V ∪ {S}
            Eliminate from V the solution with the smallest value of eval()
            if max_{S1,S2∈V} |eval(S1) − eval(S2)| ≤ δ then
                Return best solution from V
            end if
        end for
    end for
    Return the best solution from V
end.
```

CPRgen has the time complexity of $O(\frac{card(v)^2}{k^2} \cdot N^2)$, while its space complexity is $O(\frac{card(v)^2}{k})$. Note that we can exit the loop in CPRgen if either no significant mutation is possible or if the maximal number of iterations is reached. In all experiments we ran, CPRgen always terminated for the first reason because the maximum number of iterations selected was quite large.

5.6 Summary Extension Algorithm (SEA)

Both CPRdyn and CPRgen algorithms start with random summaries and try to improve them by making random block substitutions. While this approach produces a summary that is better than the initial ones, it does not guarantee that this summary will be optimal. Additionally, both CPRdyn and CPRgen use Der to find candidate blocks, which may lead to the unfounded inclusion of blocks into the summary. Thus, the results of these two algorithms may fall far from the optimum.

Let us now consider the *Summary Extension Algorithm* (SEA) that searches optimal k-summaries by enumeration, like the CPRopt algorithm, but restricts the search space by applying some heuristics. Additionally, the SEA algorithm treats any insum(b) atom as the requirement for b to be included into the summary (as opposed to being *considered* for the summary in Der-based algorithms). Given a summary S, let us define the *valid summary extension*, or a set of candidate blocks that can be

added to this summary:

Definition 5.6.1 (Valid Summary Extension) *Let R be a rule set, and S be a satisfactory summary. Then the* valid summary extension *is*

$$\mathsf{VSE}_R(S) = \{\langle b, \lambda(R, b, S)\rangle \mid \lambda(R, b, S) > 0\}.$$

In other words, $\mathsf{VSE}_R(S)$ is the set of all block-coverage pairs that satisfy R with respect to the summary S. Here is an algorithm to compute $\mathsf{VSE}_R(S)$:

```
Algorithm VSE(R,S)
    R is the set of rules
    S is the current summary
begin
    S' := ∅
    for each video block b
        for each rule r ∈ R such that head(r) = insum(X)
            // Compute λ(r, b, S) and add b to the result if λ(r, b, S) > 0.
            p_out := ChooseVars(r, {X/⟨b, 1⟩}, S)
            if p_out > 0 then S' := S' ∪ {⟨b, p_out⟩}
        end for
    end for
    return S'
end.

Algorithm ChooseVars(r,θ,S)
    r is the rule
    θ is the variable assignment set
    S is the current summary
begin
    if exists variable X ∈ r such that X ∉ θ then
        p_out := 0
        for each BCP ⟨b, p⟩ ∈ S
            // Assign one more variable and recurse, maximizing p_out.
            p' := ChooseVars(r, θ ∪ {X/⟨b, p⟩}, S)
            if p' > p_out then p_out := p'
        end for
    else
        // Substitute variables and compute p_out.
        p_out := min_{a∈body(r)} λ(aθ, S)
    end if
    // Return λ(r, b, S).
    return p_out
end.
```

The $VSE()$ algorithm iterates over all rules in R and all blocks in a video looking for blocks that satisfy R. $VSE()$ uses the $ChooseVars()$ algorithm to compute $\lambda(r, b, S)$ for each block b and rule r, and adds $\langle b, \lambda(r, b, S)\rangle$ to the output if $\lambda(r, b, S) > 0$. As BCP *set union* is used to add new block-coverage pairs to the output, pairs with lower coverages are automatically replaced with higher coverage pairs.

Suppose we start with some rule set R and an empty summary $S = \emptyset$ that *is* satisfactory w.r.t. R. $S' = \mathsf{VSE}_R(\emptyset)$ will contain all assignments satisfying the rules whose bodies are free of membership atoms. Satisfaction of such "self-supporting" rules does not require any blocks to be in the summary. Notice that S' is *always* going to be satisfactory w.r.t. R and it is always true that $\forall\langle b, p\rangle \in S : \exists\langle b, p'\rangle \in S' : p' \geq p$.

We can continue applying the $\mathsf{VSE}()$ operator to S' until it stops growing. At each stage, the union of S and any subset of S' has all the properties of a satisfactory summary:

$$\forall V \subseteq \mathsf{VSE}_R(S) : \ S \cup V \ is\ a\ satisfactory\ summary.$$

I now present a recursive algorithm to find the BCP set corresponding to the best summary.

```
Algorithm BestSummary(R,S,l,N)
   R is the set of rules
   S is the initial summary
   l is the maximal summary length
   Q is a sorted list of up to N summaries
begin
   Q := ∅
   S' := VSE_R(S)
   // Remove assignments already present in S.
   for each BCP ⟨b,p'⟩ ∈ S' such that ∃⟨b,p⟩ ∈ S
      if p' > p then S := S ∪ {⟨b,p'⟩}
      S' := S' − {⟨b,p'⟩}
   end for
   // Find N best summaries...
   for each BCP set V ⊆ S' such that length(V ∪ S) ≤ l
```

```
        Q.add(V ∪ S, eval(V ∪ S))
        if size(Q) > N then Q.delete(tail(Q))
    end for
    // ...and try to grow them.
    if Q = ∅ then BestS := S
    else
        BestS := head(Q)
        for each summary V ∈ Q
            V' := BestSummary(R, V, l, N)
            if eval(V') > eval(BestS) then BestS := V'
        end for
    end if
    return BestS
end.
```

The $BestSummary()$ is a greedy breadth-first search algorithm with the branching factor limited to N. The quality of summaries in this algorithm is measured in terms of the $eval()$ function that can be computed in various different ways, depending on what the user deems important.

5.6.1 Improving SEA Algorithm

One can easily see by examining the $BestSummary()$ algorithm that it is very time-consuming. After all, by calling $VSE()$, it enumerates all the possible variable assignments for each rule and then enumerates all the subsets of the $\mathsf{VSE}_R(S)$. Thus, the next question is: How can these algorithms be improved?

Let us look at how $\mathsf{VSE}_R(S)$ is computed. Suppose we are considering adding a block b to a summary S that already has $\langle b, p_{min} \rangle \in S$. Given a variable assignment θ, compute the *assignment coverage* $p_{ass} = min_{X/\langle b,p \rangle \in \theta} p$. It follows from the interpretation definition that $\lambda(r, b, S) \leq p_{ass}$. Applied to the $ChooseVars()$ algorithm, it means that $p_{out} \leq p_{ass}$. On the other hand, by the definition of the BCP set union, to update b's membership in the summary, the existing membership must have $p_{out} > p_{min}$. Therefore, the variable assignments under consideration can be limited

to the ones with $p_{ass} > p_{min}$:

```
Algorithm VSE'(R,S)
    R is the set of rules
    S is the current summary
begin
    S' := ∅
    for each video block b
        // Compute lower bound on λ(r, b, S).
        if ⟨b, p⟩ ∈ S then p_min := p else p_min := 0
        for each rule r ∈ R such that head(r) = insum(X)
            // Compute λ(r, b, S) and add b to the result if λ(r, b, S) > p_min.
            p_out := ChooseVars'(r, {X/⟨b, 1⟩}, S, p_min)
            if p_out > p_min then
                S' := S' ∪ {⟨b, p_out⟩}
                p_min := p_out
            end if
        end for
    end for
    return S'
end.

Algorithm ChooseVars'(r,θ,S,p_min)
    r is the rule
    θ is the variable assignment set
    S is the current summary
    p_min is the lower bound on assignment coverages
begin
    if exists variable X ∈ r such that X ∉ θ then
        p_out := p_min
        for each BCP ⟨b, p⟩ ∈ S such that p > p_out
            // Assign one more variable and recurse, maximizing p_out.
            p' := ChooseVars'(r, θ ∪ {X/⟨b, p⟩}, S, p_out)
            if p' > p_out then p_out := p'
        end for
    else
        // Substitute variables and compute p_out.
        p_out := min_{a∈body(r)} λ(aθ, S)
    end if
    // Return λ(r, b, S).
    return p_out
end.
```

In the $VSE'()$ algorithm, we compute the lower bound on $\lambda(r, b, S)$ and pass it to the $ChooseVars'()$ algorithm as p_{min}. As $ChooseVars'()$ computes $\lambda(r, b, S)$, it constantly updates the bound and skips variable assignments falling below the bound. Finally, $ChooseVars'()$ returns computed $\lambda(r, b, S)$ to $VSE'()$. If this value is bigger

than p_{min}, $VSE'()$ adds $\langle b, \lambda(r, b, S)\rangle$ to the result.

Furthermore, to avoid extra recursions, each variable can be substituted immediately as its assignment is chosen and the atoms can be evaluated as they become ground:

```
Algorithm VSE"(R,S)
    R is the set of rules
    S is the current summary
begin
    S' := ∅
    for each video block b
        // Compute lower bound on λ(r, b, S).
        if ⟨b, p⟩ ∈ S then p_min := p else p_min := 0
        for each rule r ∈ R such that head(r) = insum(X)
            // Evaluate atoms that involve only X.
            p_cur := 1
            for each feature atom a ∈ body(r) such that X ∈ a
                p' := λ(a[X/⟨b, 1⟩], S)
                p' < p_cur then p_cur := p'
            end for
            // Compute λ(r, b, S) and add b to the result if λ(r, b, S) > p_min.
            if p_cur > p_min then
                p_out := ChooseVars''(r[X/⟨b, 1⟩], S, p_cur, p_min)
                if p_out > p_min then
                    S' := S' ∪ {⟨b, p_out⟩}
                    p_min := p_out
                end if
            end if
        end for
    end for
    return S'
end.

Algorithm ChooseVars"(r,S,p_cur,p_min)
    r is the rule
    S is the current summary
    p_cur is the current coverage
    p_min is the lower bound on p_cur
begin
    if doesn't exist variable X ∈ r then
        return p_cur
    else        p_out := p_min
        for each BCP ⟨b, p⟩ ∈ S such that p > p_out
            // Compute new p_cur w.r.t. assignment X/⟨b, p⟩.
            p'_cur := p_cur
            for each atom a ∈ body(r) such that X ∈ a and a[X/⟨b, p⟩] is ground
                p' := λ(a[X/⟨b, p⟩], S)
                if p' < p'_cur then p'_cur := p'
```

```
            end for
            // Continue substitution until we hit the lower bound or run out of variables.
            if p'_cur > p_out then
                p' := ChooseVars''(r[X/⟨b, p⟩], S, p'_cur, p_out)
                if p' > p_out then p_out := p'
            end if
        end for
        // Return λ(r, b, S).
        return p_out
    end if
end.
```

This version of the algorithm is based on the observation that for any ground atom $a \in r$, $\lambda(r, b, S) \leq \lambda(a, S)$. Therefore, $VSE''()$ starts by evaluating all atoms that become ground after substituting the head variable. If the resulting p_{cur} value is higher than the lower bound, $VSE''()$ calls $ChooseVars''()$ to instantiate and evaluate other atoms.

5.7 Priority Curve Algorithm (PCA)

All algorithms previously described in this chapter relied on the explicitly defined objective function $eval()$ whose value determined the quality of each summary. In this section, I am going to present an alternative summary creation process that does not use the objective function but relies on some implicit assumptions about CPR criteria instead. We have called this process the Priority Curve Algorithm (PCA), not to be mistaken for the Principal Component Analysis (PCA) often used for the data dimensionality reduction. The PCA algorithm [4], shown in Figure 5.3 consists of the following main steps.

We first split video into blocks. This can be done ether by assigning to each block an equal number of frames (let us say 1800 frames, or a minute of 30fps video) or by using any of the standard video segmentation algorithms [32, 38, 48, 79].

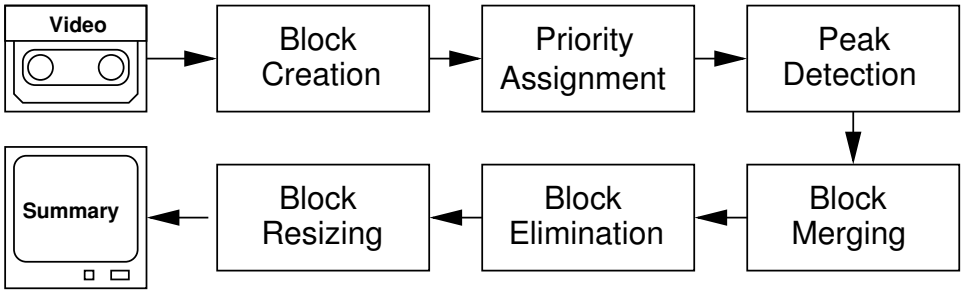


Figure 5.3: PCA Architecture.

The blocks are then fed to the *priority assignment* stage where content of each block is examined and assigned a priority. The priority assignment can be implemented both by human annotation and some image processing algorithms. The prototype system implemented by our team in Naples, for example, uses such algorithms to detect goal shots and red card events in soccer videos.

The prioritized blocks numbered in the order of increasing timestamps are then sent to the *peak detection* stage. To understand how this stage works, let us arrange all blocks on a graph with time counted along the horizontal axis and block priority plotted along the vertical axis. The resulting *priority curve* is going to have *peaks*, as shown in Figure 5.4. These peaks corresponding to high-priority blocks are identified at the peak detection stage.

The priority curve with detected peaks is then shipped to the *block merging* stage. Intuitively, peaks that are close to each other correspond to the same or similar events even though they have been assigned to different blocks by the video segmentation algorithm. Therefore, we analyze blocks corresponding to nearby peaks for similarity and merge similar blocks. The merged block's priority becomes the sum of all original blocks' priorities.

Next, merged blocks are passed through the *block elimination* stage which uses

some statistical rules to eliminate low-priority blocks. For instance, if the average priority is μ and the standard deviation is σ, we may want to eliminate all blocks whose priorities are less than $\mu - 3\sigma$, as the classical statistical model says that most objects in a normal distribution must occur within 3 standard deviations of the mean. Other statistical rules can be used as well.

Finally, the remaining blocks are delivered to the *block resizing* stage that crops them to fit the desired summary length. Each block is allocated a number of frames in the summary proportional to its priority. Blocks that fit their allocations are left untouched, while blocks that exceed their allocations are cropped to fit their designated numbers of frames.

Let us now take a more detailed look at some PCA components.

5.7.1 Peak Identification Module

Let $b_1, \ldots, b_n$ be the blocks in the video (e.g. after the segmentation process). Let p_i denote the priority of block b_i.

Definition 5.7.1 ((r,s)-peak) *Suppose $r \in (0, n/2]$ is an integer and $s \in [0,1]$ is a real number. Blocks $b_j, \ldots, b_{j+r}$ are said to be an (r,s)-*peak *iff*

$$\frac{\sum_{j \leq i \leq j+r} p_i}{\sum_{j-\frac{r}{2} \leq i \leq j+\frac{3r}{2}} p_i} \geq s$$

Suppose we wish to check if a sequence of r blocks $S_1 = b_j, \ldots, b_{j+r}$, constitutes a peak. The above definition looks at $\frac{r}{2}$ blocks before the sequence as well as $\frac{r}{2}$ blocks after the sequence, i.e. the sequence $S_2 = b_{ji\frac{r}{2}}, \ldots, b_j, \ldots, b_{j+r}, \ldots, b_{j+\frac{3r}{2}}$ is considered. This latter sequence S_2 is of width $2r$. We sum up the priorities of all blocks in S_2 - let us call this sum s_2. Likewise, we sum up the priorities of all blocks in S_1 and call this priority s_1. Clearly, $s_1 \leq s_2$. If $\frac{s_1}{s_2}$ exceeds or equals s, then we

decide that the contribution of the priorities of the peaks in S_1 is much larger than that in S_2 and so S_1 constitutes a peak. It is important to note that r and s must be chosen by the application developer.

Figure 5.4 shows two examples of peaks corresponding to r, s values of $(6, 0.65)$ and $(4, 0.6)$ respectively. Dotted rectangles signify peaks, with s-values shown for the most significant peaks. As seen from the figure, peaks often occur in clusters. While the upper graph corresponds to wide ($r = 6$) peaks, parameters in the lower graph allow for narrower ($r = 4$) and slightly lower ($s = 0.6$ as opposed to $s = 0.65$) peaks. As result, the lower graph contains more peaks and smaller clusters.

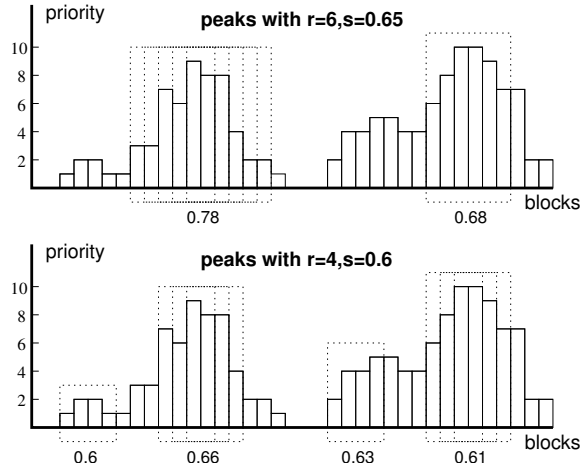


Figure 5.4: Peaks.

Here is a simple algorithm that, given a sequence of video blocks and r, s values, will find all blocks that belong to (r, s)-peaks:

Algorithm Peaks(v,r,s)
  v is a sequence of block-priority pairs
  r is the peak width
  s is the peak height
begin
  $Res := \emptyset$
  for each $j \in [r, card(v) - r]$ do
    $center := 0$
    $total := 0$

```
        for each ⟨b_i, p_i⟩ ∈ v such that i ∈ (j − r, j + r] do
            total := total + p_i
        end for
        for each ⟨b_i, p_i⟩ ∈ v such that i ∈ (j − r/2, j + r/2] do
            center := center + p_i
        end for
        if center/total ≥ s then
            Res := Res ∪ {⟨b_i, p_i⟩ ∈ v | i ∈ (j − r/2, j + r/2]}
        end if
    end for
    return Res
end
```

The $Peaks()$ algorithm slides a $2r$-wide window along a sequence of blocks, computing the total sum of block priorities in that window ($total$). It then computes the sum of block priorities in a narrower r-wide window in the middle of the $2r$-wide window ($center$), as shown in Figure 5.5. When the ratio of these two sums $\frac{center}{total}$ exceeds the threshold s, all blocks in the r-wide window are picked as a *peak*.

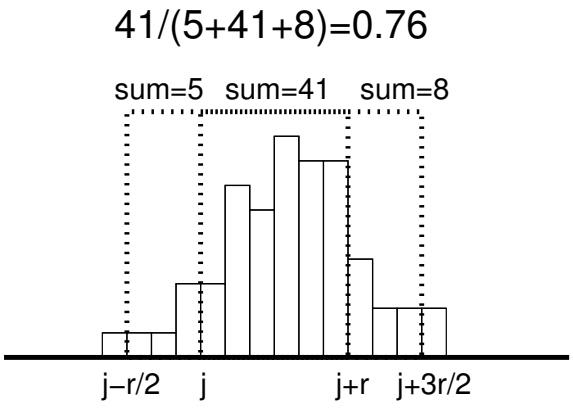


Figure 5.5: Peaks() Algorithm Analyzing a Peak.

Example 5.7.1 *Consider the very small fragment shown in Figure 5.5. At some time, the $Peaks()$ algorithm will focus its window of length $2r$ on the segment from $j - \frac{r}{2}$ to $j + \frac{3r}{2}$ shown in the figure. It will compute the sum of the priorities of the blocks in the entire window of length $2r$ (which is $5 + 41 + 8 = 54$) as well as the sum of*

the priorities of the window of length r in the center of the window of length $2r$ - the priority there is 41. As a consequence, the ratio of these is $\frac{41}{54} = 0.76$. If 0.76 exceeds the s that the user has picked, then the sequence of blocks from j to $j+r$ is considered a peak.

Note that the performance of the $Peaks()$ algorithm can be improved by avoiding computation of $center$ and $total$ iteratively in each iteration of the outer loop. After the first iteration of the outer loop, these values can be updated in constant time. This optimization is not included in the above algorithm as it complicates the simplicity of the algorithm, but it is easy to incorporate. The $Peaks()$ algorithm has complexity of $O(r \cdot card(v))$, being linear with respect to the number of input blocks.

5.7.2 Block Merging Module

The peak identification algorithm eliminates all blocks that are not (r, s)-peaks for the r, s values selected by the application developer. Let $Peaks(v, r, s)$ be the set of all blocks from the original video that contain peaks. Consider a set $\{(b_i, b_{i+1}) \,|\, b_i, b_{i+1} \in Peaks(v)\}$ of all pairs of blocks that are adjacent to each other. In general when adjacent blocks are peaks, they may describe the same event. The main goal of the block merging module is to merge adjacent blocks that may be very similar, so that repeating blocks can be treated as a single block in the later processing steps (such as resizing).

A *block similarity function* is a function sim that takes two blocks as input and returns a non negative real number as output. The smaller the number returned, the more similar the blocks are considered to be. There are many ways in which we could implement block similarity functions. Here are a few examples:

1. sim_{diff}: We could use any classical image differencing algorithm $idiff$ [35] to return the similarity between two frames and we could set the similarity between the two blocks to be the similarity between the two most similar frames, drawn from each block.

2. sim_{text}: In the event that the videos in question have an accompanying text transcript, we could identify the text blurb associated with each of the two blocks and set the similarity of the two blocks to be equal to the similarity between the two text transcripts using any classical method to evaluate similarities between text documents.

3. sim_{vec}: As is often common in image processing, we could associate a color and/or texture histogram with each block and return the similarities between the histograms using root mean squared distance or the L_1 metric [78].

To simplify the block merging process, let us assume that $Peaks(v, r, s)$ returns a set of block-priority pairs of the form $\langle b_i, p_i \rangle$, as opposed to a set of blocks, and adjacent blocks can be concatenated with the $\oplus$ operator. The block merging algorithm then takes as input, *any* block similarity function between blocks, together with a set of block-priority pairs, and returns a new set of merged blocks-priority pairs, as follows.

```
Algorithm Merge(v,sim(),d)
    v is a sequence of block-priority pairs
    sim() is a similarity function on blocks
    d is the merging threshold
begin
    Res := ∅
    B := first block-priority pair ⟨b_1, p_1⟩ ∈ v
    for each ⟨b_j, p_j⟩, ⟨b_{j+1}, p_{j+1}⟩ ∈ v do
        if sim(b_j, b_{j+1}) ≥ d then
            B := ⟨B.b ⊕ b_{j+1}, B.p + p_{j+1}⟩
        else
            add B to the tail of Res
            B := ⟨b_{j+1}, p_{j+1}⟩
        end
```

```
    end for
    add B to the tail of Res
    return Res
end
```

The $Merge()$ algorithm considers all pairs of blocks b_j, b_{j+1}, concatenating them together into a bigger block $B.b$, as long as $sim(b_j, b_{j+1})$ value stays above the threshold d. The priority $B.p$ of the newly merged block is computed as a sum of individual priorities of its parts. The $Merge()$ algorithm has linear complexity with respect to the number of blocks in its input.

5.7.3 Block Elimination Module

Suppose S is the set of blocks from the original video after the block merging step has been applied to the set of blocks in $Peaks(v, r, s)$. In the block elimination module, we would like to remove from this set all blocks whose priorities are less than a certain threshold. Let us do it by computing the mean μ and the standard deviation σ for all priorities in S. Now, given an integer $m \geq 0$, let us define a function $Drop(S, m)$ that drops from S all blocks whose priorities are less than $\mu - m\sigma$. $Drop()$ can be easily implemented by iterating over all blocks returned by the $Merge()$ algorithm. Thus, the result of $Drop(Merge(Peaks(v, r, s), d), m)$ will be a set of all high-priority merged peaks taken from v, with respect to the r, s, d, m parameters.

5.7.4 Block Resizing Module

Even after eliminating some low-priority blocks in the previous step, the total frame count of the remaining blocks may still exceed the limit k imposed in the beginning of this paper. In such a case, we have to truncate some blocks to fit the limit. Clearly, blocks with higher priorities must have more prominence in the summary and thus

occupy a larger percentage of frames. We then devise an algorithm that allocates to each block a number of frames proportional to its priority and truncates blocks to fit the limit of k frames.

```
Algorithm Resize(v,k)
    v is a sequence of block-priority pairs
    k is the desired summary length
begin
    Res := ∅
    p_total := Σ_{⟨b,p⟩∈v} p
    p' := 0
    k' := 0
    for each ⟨b,p⟩ ∈ v do
        if len(b) ≤ p·k / p_total then
            Res := Res ∪ {⟨b,p⟩}
            v := v \ ⟨b,p⟩
            p' := p' + p
            k' := k' + len(b)
        end if
    end for
    p_total := p_total − p'
    k := k − k'
    for each ⟨b,p⟩ ∈ v do
        alloc := p·k / p_total
        b' := b truncated to alloc frames
        Res := Res ∪ {⟨b',p⟩}
    end for
    return Res
end
```

5.8 Implementation and Experiments

Our experiments were conducted on a prototype summarization system developed in Java for the Windows 2000 platform. Oracle8i and MS Access database backends were used to store and access video data. The system consists of approximately 2500 lines of code.

The experiments used a collection of 50 soccer videos in AVI format captured at 30fps in 640×480 resolution. Videos varied in length from 90 to 120 minutes and

had at least 162000 frames each. The feature extraction was done both manually and by an image processing algorithm that detected goals, so that each video had about 40 annotated blocks on average. Features described players and actions occuring in a block, such as “goal”, “cross”, “offside”, “celebration”, or “shot”.

To assess the quality of summaries produced by our summarization algorithms, we used them to create 2, 4, and 6 minute summaries of all 50 videos. A group of approximately 200 students at the University of Naples evaluated the quality of resulting summaries by comparing them to original videos. In order to rate the quality, students gave each summary a rank from A (excellent) to E (unacceptable). Given the high subjectivity of this method, the results below have to be taken with a grain of salt, but it has been shown that they remain consistent when the experimental conditions (CPR weights, summary length, etc.) are changed.

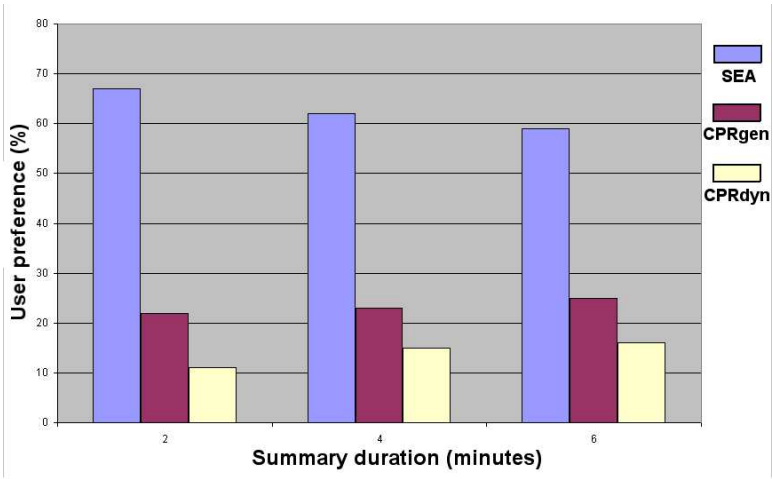


Figure 5.6: Summary Quality Comparison (without PCA).

Figure 5.6 shows students’ preferences for the summaries produced by the first three algorithms. SEA algorithm was deemed to produce the best results in 67% of the cases. Furthermore, 81% of the participants gave the SEA algorithm an A during

that test run.

Figure 5.7 shows students' preferences for the summaries produced by all four algorithms. The PCA algorithm has been deemed to produce best results in 46% of cases in this run, followed by the SEA algorithm.

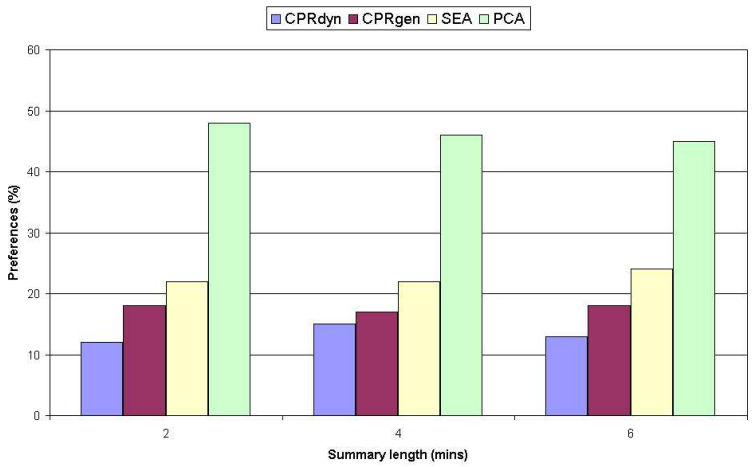


Figure 5.7: Summary Quality Comparison (with PCA).

In order to understand how changes in the video content specification affect the degree of user satisfaction, we produced several summaries of the same video using four different content specifications. Figure 5.8 shows user preferences for each of these four cases. We have found that, *once informed on the objective of the summarization*, i.e. which actions/events in the summary had been considered more important by the specification, users expressed preferences similar to the previously discussed results.

Finally, we assessed the performance of the four algorithms using a Pentium3 800MHz machine with 128MB SDRAM. Figure 5.9 shows the results, with the PCA algorithm outperforming the other three algorithms.

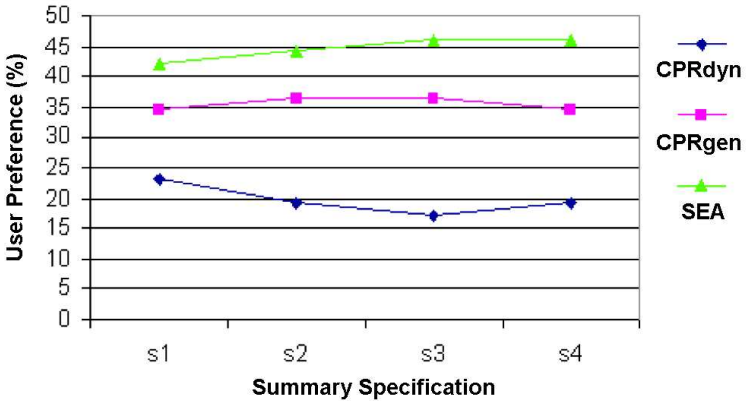


Figure 5.8: Summary Quality Variation.

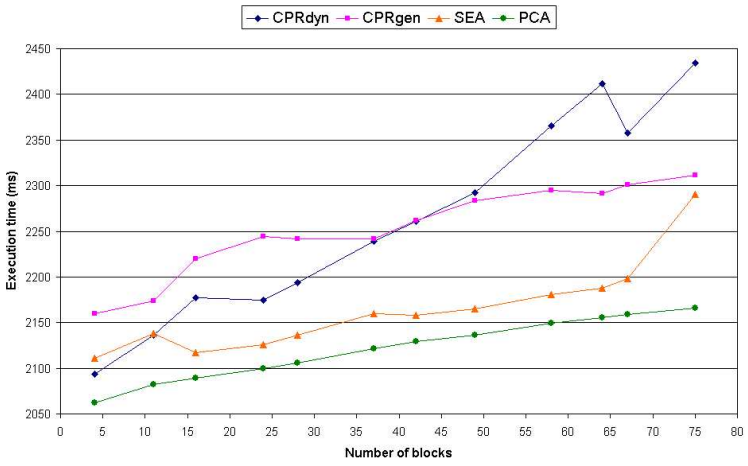


Figure 5.9: Algorithms' Performance.

5.9 Related Work

Over the past years, there was a variety of efforts in automatic video skimming and summarization while searching for certain faces, narrative, or other features. Nevertheless, the idea of creating coherent summaries based on the user specifications has not received much attention.

A work that specifically targets creation of video summaries has been done by He

et. al. at Microsoft [39]. The goal of their project was to summarize presentation videos that were accompanied by the PowerPoint slides being presented. The system used three main criteria to determine the relative importance of video segments: (i) the moments when slides were changed, (ii) lecturer's voice pitch, and (iii) users' interest in different parts of the presentation as they skipped through it. The researchers have invested a lot of effort into estimating how relevant and coherent automatically generated summaries were in comparison to their author-generated counterparts. The obvious peculiarity of this work is that each video sequence was synchronized to the actual PowerPoint presentation, which is not the case with most real life videos. The video content itself has not been analyzed in any way, as it was mainly limited to the narrator's talking head. Also, although there is some implicit user feedback present (user access statistics were used during summary creation), the users have no control over what will be summarized.

Another project in video summarization was undertaken by DeMenthon, Kobla, and Doermann at UMD [18] who represented a changing vector of frame features (such as overall macroblock luminances) with a multi-dimensional curve and applied a curve simplification algorithm to select "key" frames. The selected frames then constitute a summary. The algorithm allows to control the size of a summary by choosing the degree of simplification.

The Informedia project by Kanade and Smith [44] selects frames from documentaries and news bulletins by detecting important words in the accompanying audio. The MoCA project by Lienhart et. al. [50] composes film previews by picking special events, such as zooming of actors, explosions, shots, etc. There are many other systems that use various video characteristics to detect key frames and compose summaries out of these frames [34, 28].

While the key frame approach has proven to work rather well, it looks at certain features of a video instead of considering its semantic content. Thus, users have no direct control over the summary content and quality. The CPR approach used by our group allows for such control. The idea of creting summaries by maximizing an objective function has been proposed by Yahiaoui, Merialdo et. al. [86]. Our work, covered in this chapter, uses the objective function idea but provides a more general framework that takes into account users' preferences about the summary content and quality.

5.10 Conclusion

Our team has developed a theoretical model for the video summarization and shown several algorithms for creating summaries. The proposed model for video summarization is the first that takes into account both semantic content and the user preferences.

We have futher implemented algorithms, and conducted experiments to analyze their performance. The results clearly show benefits of the Priority Curve Algorithm, followed by the Summary Extension Algorithm.

Chapter 6

Document Summarization With Stories

6.1 Introduction

On many life occasions, one needs to find exhaustive information about a particular person, a place, an event, or an artefact. For instance, a tourist wandering through the archaeological museum at Pompeii and encountering a painting titled "Death of Pentheus" may want to know who pictured characters are, when and how this painting has been made, and who the painter is. At the same time, but in a completely different setting, a soldier guarding a checkpoint would like to know the background of people who come by his checkpoint, while a military truck driver needs to know the history of bombings, ambushes, and other accidents that have occurred along his travel route.

While each person in the above examples needs information from a different domain, in all instances we are basically interested in a *story* behind a given subject. Historically, people referred to encyclopedias, dictionaries, and other reference documents for such stories. Of course, lookup of printed sources is a time consuming process, especially when multiple documents have to be consulted. It is also difficult to update printed information, not to mention the fact that it can't be easily consulted

in the field, as our examples require.

In the recent years, the World Wide Web has taken the role of the main reference source for many people. Traditional encyclopedias and dictionaries, as well as news outlets, have "migrated" to the Web, providing instant access, up-to-date information, and search capabilities. Nevertheless, some problems persist:

- Finding and consulting Web sources is faster than a traditional library search but it still takes time.
- Making sense of multiple sources takes still more time and analytical experience, just like the analysis of printed documents.
- Information obtained from the Web is often unreliable or inconsistent.
- In many applications (such as military examples given above) there are proprietary sources of information, such as databases, intelligence reports, or internal newsfeeds that also have to be consulted.

Thus, an automated tool that solves the above problems for the user and presents him with a cohesive narrated story about the subject of his interest would be very helpful for many people.

Let us now return to our examples. Notice that stories requested in them are dramatically different. In the case of Pompeii, tourists may be interested in cultural, historical, mythological, and artistic aspects of subjects. On the other hand, these aspects are of no interest to soldiers who mainly focus on threat assessment. Thus, the content of a story depends both on the available facts and the domain of user's interests.

There are two other important aspects of stories: they must be *succinct* and they must allow the user to *explore* different facets of the story that are of interest to him or

her. So, a tourist in the museum example may want to find out more about painting's author and then about the place where painter has lived.

In this chapter, I present a framework for creating stories about given subjects based on the information extracted from heterogenous data sources, such as the Web, the relational databases, and so forth. Furthermore, this chapter presents several algorithms that generate an optimal story with respect to the priority of information it contains, the continuity of its narrative, and the non-repetition of covered facts. Finally, I discuss the architecture and the bimplementation of a prototype STORY system that automatically creates and delivers stories to client devices such as computers, PDAs, cell phones, and so forth.

The work covered in this chapter is partially based on the work in video summarization [24] covered in Chapter 5.

6.2 The Data Model

According to the Oxford English Dictionary (online edition), a story is *"A narrative, true or presumed to be true, relating to important events and celebrated persons of a more or less remote past; a historical relation or anecdote."* In the context of computing, a narrative becomes an interactive multimedia presentation. Such an approach allows a piece of plain text or speech to be treated as a special case of a narrative.

Returning to the painting of Pentheus' punishment and death at the House of the Vetti in Pompeii (see Figure 6.1), a visitor to Pompeii may have a number of questions to ask about this painting:

- Who was Pentheus?
- Who punished him?

Figure 6.1: Pentheus torn apart by his mother Agave and Maenads (House of the Vetti, Pompeii).

- Why he was punished?
- Was this event depicted by other artists at the same period or in earlier periods or in later periods in the same or different geographical region?
- Who is Vetti?

In order to answer the above questions and support further exploration of topics related to the initial topic of interest, let us introduce the concept of an *entity* characterized with a set of *attributes* that can be arranged into a *story*.

Definition 6.2.1 (Entity) *We assume the existence of some set $\mathcal{E}$ whose elements are called entities.*

Intuitively, entities describe the subjects of interest. In a museum, such subjects can be all the known people depicted via images or sculptures shown in the museum, as well as all people related to those people in some way. Additionally, the set of entities could include all places depicted. In the case of soldiers guarding a checkpoint, entities of interest could include all people about whom intelligence agencies

have any information, as well as various kinds of resistance and terror groups, front companies, etc. Note that there is no need to explicitly enumerate this set of entities – they could, for example, be discovered by a simple algorithm seeking proper nouns.

Definition 6.2.2 (Ordinary Attributes) *Suppose $\mathcal{E}$ is a set of entities. We assume the existence of a set $\mathcal{A}$ whose elements are called* ordinary attributes. *Each attribute A has an associated domain $dom(A)$. We say that $\mathcal{A}$ is a* set of ordinary attributes associated with the set $\mathcal{E}$ of entities *if* $\mathcal{E} \subseteq \bigcup_{A \in \mathcal{A}} dom(A)$.

The above requirement merely ensures that each entity can be characterized by the values of ordinary attributes.

The story about Pentheus may have many ordinary attributes. One such attribute might be `mother`, with the domain being the set of all alphabetical strings. The *value* of this attribute w.r.t. the painting of Pentheus shown in Figure 6.1 could be the string "Agave". An attribute `persons` could have as its domain, the powerset of the set of people known in Greek mythology. In the Pentheus example, the value of this attribute could be $\{Pentheus, Agave, Maenads\}$ – note that Maenads is not one person, but rather a collective name for a group of people.

Notice that the value of the `mother` attribute may be an entity by itself. In the Pentheus example, the entity Agave (his mother) may have its own attributes, so that a new story could be created about her. However, if Agave is *not* an entity then there is definitely no story about her.

Definition 6.2.3 (Generalizing Ordinary Attributes) *Suppose A is an ordinary attribute. Then a* generalization function *for attribute A is a mapping Γ_A from $2^{dom(A)}$ to $dom(A)$.*

For example, suppose we have an attribute called `occupation` whose domain is the set of all strings. A generalization function Γ_{occ} may map a set of strings of the form "King of . . ." to a single string "king" and "Bishop of . . ." to "bishop". This is just one example of a generalization function – many more are possible.

Figure 6.2: Pope Paul III and his nephews Alessandro and Ottavio Farnese (Capodimonte Museum, Naples).

In the above discussion, attributes had constant values. However, in many situations attribute values may change with time. For example, Pope Paul III (shown in a painting in Figure 6.2) may have an `occupation` attribute with the value "cardinal" for a time period from 1493 to 1533 and "Pope" from 1534 to 1549. In order to express such values, let us introduce the concept of a time-varying attributes.

Definition 6.2.4 (Time Varying Attribute) *A* time-varying attribute *is a pair* $(A, dom(A))$ *where* A *is the name of the attribute and* $dom(A)$ *is the domain of values for the attribute.*

Definition 6.2.5 (Timevalue) *A* timevalue *for a time-varying attribute* $(A, dom(A))$ *is a set of triples* (v_i, L_i, U_i) *where* $v_i \in dom(A)$ *and* L_i, U_i *are either integers or the special symbol* $\perp$ *(denoting unknown). A timevalue is* fully specified *iff there is no triple of either the form* $(v_1, \perp, U_i)$ *or* $(v_i, L_i, \perp)$ *or* $(v_i, \perp, \perp)$ *in it.*

Intuitively, if an object has a time-varying attribute A with a timevalue of $\{(v_1, 15, 20), (v_2, 25, 30)\}$ then A has value v_1 in a time period from 15 to 20 and value v_2 in a time period from 25 to 30. In the case of Pope Paul III, the timevalue of `occupation` is given by $\{(Pope, 1534, 1549), (Cardinal, 1493, 1533)\}$.

Note 6.2.1 *Though one may have allowed timevalues to be more expressive, the reader will see later (Note 6.3.1) that doing so causes a huge additional burden on the system implementor.*

Definition 6.2.6 (Consistent Timevalue) *A timevalue* tv *for a time-varying attribute* $(A, dom(A))$ *is* consistent *iff there is no pair* $(v_1, L_1, U_1), (v_2, L_2, U_2)$ *in* tv *such that* $v_1 \neq v_2$ *and* $L_1, U_1, L_2, U_2 \neq \perp$ *and such that the intervals* $[L_1, U_1] \cap [L_2, U_2]$ *intersect.*

The consistency of a timevalue ensures that the attribute does not have two distinct values at the same time (e.g. Pope Paul III has never been both Pope and a cardinal at the same time). Thus, the timevalue $\{(Pope, 1534, 1549), (Cardinal, 1493, 1533)\}$ for Pope Paul III's occupation attribute is consistent.

Note, however, that had we wanted to allow a person to have multiple occupations at the same time, we could simply define the domain of `occupation` to be the *powerset* of all strings rather than the set of all strings.

Note 6.2.2 *Throughout the rest of this chapter, I will abuse notation and use the term "attribute" to refer to both ordinary and time-varying attributes. The context will determine the usage.*

Just like ordinary attributes, the time-varying attributes can be generalized.

Definition 6.2.7 (Generalizing Time Varying Attributes) *Suppose* $(A, dom(A))$ *is a time-varying attribute. A* generalization function *for* A *is a mapping* Γ_A *from a timevalue for attribute* $(A, dom(A))$ *to a singleton timevalue for attribute* $(A, dom(A))$ *such that if* $\Gamma_A(X) = \{(v, L, U)\}$ *then*

$$[L, U] \subseteq [min_{(v_i, L_i, U_i) \in X} L_i, max_{(v_i, L_i, U_i) \in X} U_i] \ \wedge \ \exists (v_j, L_j, U_j) \in X \ [L, U] \cap [L_j, U_j] \neq \emptyset.$$

For example, a generalization function may map the set of timevalues { ("Bishop of Massa", 1538, 1552), ("Bishop of Nice", 1533, 1535) } to a singleton timevalue { ("bishop", 1533, 1552) }.

Note that if the definition of generalization function for time-varying attributes did not require that $\exists (v_j, L_j, U_j) \in X [L, U] \cap [L_j, U_j] \neq \emptyset$ then we would have a problem. The generalization function could, for example, return {("bishop",1536,1537)} even though the original input said nothing about the time interval $[1536, 1537]$.

Definition 6.2.8 (Story Schema) *A* story schema *consists of a pair* $(\mathcal{E}, \mathcal{A})$ *where* $\mathcal{E}$ *is a set of entities and* $\mathcal{A}$ *is a set of attributes associated with* $\mathcal{E}$*. I will use* $\mathcal{A}^o$ *(resp.* $\mathcal{A}^{tv}$*) to denote the ordinary (resp. time-varying) attributes in* $\mathcal{A}$*.*

Returning to our example of the Pompeii archaeological site, the set of entities of interest to tourists could be defined as the union of (i) all artefacts in Pompeii that are of interest (including paintings, sculptures, etc), (ii) all objects and events depicted by those artefacts, and (iii) any people, places, and events related to entities in the

previous two categories. While the first two categories can be determined manually, based on the museum inventory, the last category is very vast and thus asks for an automatic retrieval.

Definition 6.2.9 (Instance) *An* instance *w.r.t. story schema* $(\mathcal{E}, \mathcal{A})$ *is a partial mapping* $\mathcal{I}$ *which takes an entity* $e \in \mathcal{E}$ *and an attribute* $A \in \mathcal{A}$ *and returns as output, a value* $v \in dom(A)$ *when* A *is an ordinary attribute, and a timevalue* $\{(v, L, U) \mid v \in dom(A)\}$ *when* A *is a time-varying attribute.*

Let us use the notation $\mathcal{I}(e, A) = \perp$ to indicate that $\mathcal{I}(e, A)$ is undefined. Here is a couple of instance examples in the context of paintings shown in Figure 6.1 and Figure 6.2.

Example 6.2.1 (Pentheus Instance) *Pentheus was a Greek king who has made an enemy of the god Bacchus. Angered by this, the Maenads (who were priestesses worshipping Bacchus) transformed Pentheus into an animal and had his mother, Agave, kill him. A story instance may express these facts as follows:*

1. `occupation` *is a time-varying attribute with the value of* $\{(king, \perp, \perp)\}$ *which says that Pentheus has been king at an unknown time.*

2. `enemy` *is a time-varying attribute specifying the enemies of Pentheus, with the value of* $\{(\{Bacchus, Maenads\}, \perp, \perp)\}$. *Notice that Bacchus and the Maenads are also entities with their own attributes, not shown here but present in the instance.*

3. `punishment` *is a time-varying attribute specifying Pentheus' punishments, with the value of* $\{(\{$ *"transformed into an animal", "killed"* $\}, \perp, \perp)\}$.

4. `mother` *is an ordinary attribute with the value Agave.*

The table below shows attributes for other entities returned by the instance.

Entity	*Attribute*	*Value*
Pentheus	*occupation*	$\{(king, \perp, \perp)\}$
	enemy	$\{(\{Bacchus, Maenads\}, \perp, \perp)\}$
	punishment	$\{(\{\text{“}transformed\ into\ an\ animal\text{”}, \text{“}killed\text{”}\}, \perp, \perp)\}$
	mother	*Agave*
Bacchus	*occupation*	*god*
	enemy	$\{(Pentheus, \perp, \perp)\}$
	friends	$\{(Maenads, \perp, \perp)\}$
Maenads	*occupation*	$\{(priestess, \perp, \perp)\}$
	friends	$\{(Bacchus, \perp, \perp)\}$

Example 6.2.2 (Pope Paul III Instance) *The following table lists various entities occurring in Figure 6.2. Note that this example leads to possible confusion. The entity “Alessandro Farnese” shown here is not the same as Pope Paul III (who was also named Alessandro Farnese). Thus, for every attribute value, we need to specify whether it also describes an entity. I use the parenthetical comment “(v)” to denote value when not clear from context.*

Entity	*Attribute*	*Value*
Pope Paul III	*occupation*	$\{(Cardinal, 1493, 1533), (Pope, 1534, 1549)\}$
	real_name	$Alessandro\ Farnese_{(v)}$
	treaties	$(Treaty\ of\ Crespi, 1544, 1544)$
Alessandro Farnese	*occupation*	$\{(Bishop\ of\ Massa, 1538, 1552),$ $(Archbishop\ of\ Tours, 1553, 1556)\}$
Ottavio Farnese	*occupation*	$\{(Duke\ of\ Parma\ and\ Piacenza, 1547, 1586)\}$
Titian	*occupation*	$\{(painter, 1485, 1576)\}$
	teacher	$\{gentile, Giovanni\ Bellini\}$

Note that this is a very small instance – clearly a lot more is known about the individuals listed above (e.g. attributes of Titian can be the list of all paintings by him, his collaborators, etc.).

6.3 Story Computation Problem

In this section, we will look at the steps that lead to a story creation, discuss what makes one story better than the others, and finally state the story computation problem.

Let us start by defining the concepts of a *valid instance* and a *full instance* based on a set of data sources. Intuitively, these instances are used to collect all facts reported by these sources.

6.3.1 Valid and Full Instances

In order to create a story from a story schema, one may need to access a variety of sources. In the case of Pentheus, we may need to access ancient Greek texts, as well as modern works in Greek history to find out more about him. Let us assume that our data sources have an associated application program interface (this is a reasonable assumption as most commercial programs do have APIs). The *source access table* describes how to extract each attribute's value using source's API.

Definition 6.3.1 (Source Access Table) *A* source access tuple *is a structure* $(A, s, f_{A,s})$ *where* A *is an attribute name,* s *is a data source, and* $f_{A,s}$ *is a partial function (body of software code) that maps objects to values in* $dom(A)$ *when* A *is an ordinary attribute, and to timevalues over* $dom(A)$ *when* A *is a time-varying attribute. A* source access table *is a finite set of source access tuples.*

Suppose, for instance, that we want to consult the "Bacchae" (a play by Euripides) to find the value of the `mother` attribute of Pentheus. In this case, we can execute the function $f_{\texttt{mother},Bacchae}(Pentheus)$.

The source access table does not, of course, need to be populated with a function for each source and each attribute. Some sources may provide some informa-

tion, while others may not. The functions $f_{A,s}$ are *partial functions* because some sources may not have information about Pentheus. For instance, the book "Egypt" may not have any information on Pentheus, even though it supports the the function $f_{\texttt{mother},Egypt}$.

Note 6.3.1 *The application developer is responsible for specifying the functions $f_{A,s}$ in the source access table. Such a function must return timevalues when A is a time-varying attribute. This can be quite difficult. For instance, determining when Pentheus was killed from a text document is a nontrivial task. Had we allowed timevalues to be more general (e.g. to say Pentheus was killed after some other event, or to say Pentheus was killed within 5 years of yet another event), then the functions $f_{A,s}$ would need to infer this even more complex information from textual sources.*

Definition 6.3.2 (Valid Instance) *Suppose $(\mathcal{E}, \mathcal{A})$ is a story schema, SAT is a source access table, and $\mathcal{I}$ is an instance. $\mathcal{I}$ is said to be* valid *w.r.t. SAT iff for every entity $e \in \mathcal{E}$ and every attribute $A \in \mathcal{A}$, if $\mathcal{I}(e, A)$ is defined, then there is a triple of the form $(A, s, f_{A,s})$ in SAT such that $f_{A,s}(e) = \mathcal{I}(e, A)$.*

Intuitively, the above definition says that an instance is valid w.r.t. some source access table if every fact (i.e. every assignment of value to an attribute for an entity) is supported by at least one source. Note that different sources may disagree on the value of a given attribute for a given entity. For instance, one source may say Pentheus' mother is Agave, while another may say that it is Hera.

Let us now define the *full instance* that, for each entity and attribute, collects all attribute values that are supported by at least one source.

Definition 6.3.3 (Full Instance) *Suppose $(\mathcal{E}, \mathcal{A})$ is a story schema and SAT is a source access table. Suppose $\mathcal{I}$ is an instance w.r.t. $(\mathcal{E}, \mathcal{A}')$ where the attributes in $\mathcal{A}'$ are*

the same as the attributes in $\mathcal{A}$ with one difference – if an attribute $A \in \mathcal{A}$ has $dom(A) = 2^{2^S}$, then the corresponding attribute $A' \in \mathcal{A}'$ has $dom(A') = dom(A)$. Otherwise, $dom(A') = 2^{dom(A)}$, i.e. the powerset of the original domain. $\mathcal{I}$ is said to be the full instance *w.r.t. $(\mathcal{E}, \mathcal{A})$ and SAT iff for all entities $e \in \mathcal{E}$ and attributes $A \in \mathcal{A}$,*

$$\mathcal{I}(e, A) \;=\; \begin{cases} \bigcup_{\forall s\ (A,s,f_{A,s}) \in \mathsf{SAT}} f_{A,s}(e) & if\ dom(A) = 2^{2^S} \\ \{f_{A,s}(e) \mid \forall s\ (A, s, f_{A,s}) \in \mathsf{SAT}\} & otherwise \end{cases} .$$

Intuitively, the above definition says that a full instance accumulates all the facts reported by various sources, independently of whether these facts are conflicting or not. A special treatment is given to set-valued facts, whose values are merged. We will look at how conflicts between accumulated facts can be resolved later on (Definition 6.3.6).

6.3.2 Stories

Before, we have noticed that different sources may return different values for the same attribute of an entity. Some of these values can be *generalized* to produce new, presumably more concise versions. Let us then extend the story schema with appropriate tools to generalize attribute values.

Definition 6.3.4 (Generalized Story Schema) *A* generalized story schema *is a quadruple $(\mathcal{E}, \mathcal{A}, \sim, \mathcal{G})$ where $(\mathcal{E}, \mathcal{A})$ is a story schema, $\sim$ is a mapping which associates an equivalence relation on $dom(A)$ with each attribute $A \in \mathcal{A}$ and $\mathcal{G}$ is a mapping which assigns, to each attribute $A \in \mathcal{A}$, a generalization function Γ_A for attribute A.*

Thus, a *generalized* story schema adds to each attribute in the story schema an equivalence relation and a generalization function. Intuitively, an equivalence relation on

the domain $dom(A)$ of attribute A specifies when certain values in the domain are considered equivalent. For example, we may consider strings "king" and "monarch" to be equivalent. Similarly, in a time-varying attribute, we may always consider $(\text{“}king\text{”}, L, U)$ and $(\text{“}monarch\text{”}, L', U')$ to be equivalent independently of whether $L = L' \wedge U = U'$ is true or not. Likewise, in the example of Pope Paul III, the equivalence relationship may say that the triple ("Bishop of . . .",–,–) is always equivalent to other triples of the form ("Bishop of . . .",–,–) independently of whether the bishops involved governed different places.

The definition of a closed instance below takes a full instance associated with a source access table and closes it up by including all the generalized attribute values.

Definition 6.3.5 (Closed Instance) *Suppose $(\mathcal{E}, \mathcal{A}, \sim, \mathcal{G})$ is a generalized story schema and $\mathcal{I}$ is the full instance w.r.t. $(\mathcal{E}, \mathcal{A})$. The* closed instance w.r.t. a source access table SAT and generalized story schema $(\mathcal{E}, \mathcal{A}, \sim, \mathcal{G})$ *is defined as follows.*

$$\mathcal{I}'(e, A) = \mathcal{I}(e, A) \cup \{\Gamma_A(X') \mid X' \text{ is an } \sim_A \text{-equivalence class of } \mathcal{I}(e, A)\}.$$

In other words, here is how we find the closed instance associated with a given source access table and a given generalized story schema. For each entity e and each attribute A of this entity:

1. We first compute the set $\mathcal{I}(e, A)$ where $\mathcal{I}$ is the full instance associated with our source access table.

2. We then split $\mathcal{I}(e, A)$ into equivalence classes using the equivalence relation $\sim_A$ on $dom(A)$. Suppose the equivalence classes thus generated are $X_1, \ldots, X_n$.

3. For each equivalence class X_i, we compute $\Gamma_A(X_i)$ – this is the generalization of the equivalence class X_i using the generalization function Γ_A associated with attribute A. Suppose $\Gamma_A(X_i) = v_i$.

4. Finally, we insert the tuple (e, A, v_i) into the full instance.

This process is done for all entities e and all attributes A and all tuples of the form shown above are inserted into the full instance. The result is a closed instance.

A story cannot just be defined as a full instance. In the real world, the "full story" about any single person or event is likely to be very complex and involve a large amount of unimportant minutiae. For example, consider the story of Pope Paul III. Depending on what items about Pope Paul III are considered important, we may choose to merely say that he served as a bishop from 1538 to 1556 and ignore the details. However, the full instance associated with Pope Paul III may not explicitly say this – rather it might state (as in our example) that he was a bishop of this place for some time, that place for another time period, and so on. Generalization is needed to reduce these facts to a single concise fact.

Note that so far we have not tried to resolve possible conflicts between attribute values obtained from different sources. However, for the story to be consistent, these conflicts have to be resolved. In other words, suppose $\mathcal{I}'$ is a closed instance. Whenever $\mathcal{I}'(e, A)$ contains more members than prescribed by $dom(A)$, or represents an inconsistent timevalue, some mechanism is required to restore consistency by removing extra values.

Definition 6.3.6 (Conflict Management Policy) *Given an attribute A with $dom(A) = 2^{2^S}$, the* conflict management policy χ_A *is a mapping from $dom(A)$ to $dom(A)$ such that $\chi(X) \subseteq X$. For any other attribute A, χ_A is a mapping from $2^{dom(A)}$ to $dom(A)$ such that $\chi(X) \in X$.*

Following are some examples of conflict management policies.

Example 6.3.1 (Temporal Conflict Resolution) *Suppose different data sources provide different values $v_1, \ldots, v_n$ for $\mathcal{I}(e, A)$ with a value v_i being inserted into the data source at time t_i. In this case, we pick the value v_j that has been inserted the last, i.e. $t_j = max(t_1, \ldots, t_n)$. If multiple such j's exist then one is selected randomly.*

Example 6.3.2 (Source Based Conflict Resolution) *The developer of a story may assign a* credibility *c_i to each source s_i that provides a value v_i for attribute A of entity e. This strategy picks value v_i such that $c_i = max(c_1, \ldots, c_n)$. If multiple such i's exist then one is selected randomly.*

Example 6.3.3 (Voting Based Conflict Resolution) *Each distinct value v_i returned by at least one data source has a number $vote(v_i)$ which is the number of sources that return v_i. In this case, the conflict resolution strategy returns the value with the highest vote. If multiple v_i's have the same highest vote then one is selected randomly.*

These are just three example strategies. It quite easy to create hybrids of these strategies as well. For example, we could first find the values for $\mathcal{I}(e, A)$ with the highest votes and then choose the one which is most recent.

Definition 6.3.7 (Deconflicted Instance) *Suppose $(\mathcal{E}, \mathcal{A}, \sim, \mathcal{G})$ is a generalized story schema and $\mathcal{I}'$ is the closed instance w.r.t. $(\mathcal{E}, \mathcal{A})$. The* deconflicted instance *w.r.t. a source access table SAT, generalized story schema $(\mathcal{E}, \mathcal{A}, \sim, \mathcal{G})$, and conflict management policy χ is the instance $\mathcal{I}^\sharp$ such that for all entities $e \in \mathcal{E}$ and all attributes $A \in \mathcal{A}$ if $\mathcal{I}^\sharp(e, A) \neq \perp$ then $\mathcal{I}^\sharp(e, A) = \chi(\mathcal{I}'(e, A))$.*

Note that finding any arbitrary deconflicted instance is not enough. The instance $\mathcal{I}_{null}$ which is undefined for all $\mathcal{I}_{null}(e, A)$ is free of conflicts – however it is not very

useful as it has no information in it. I will later show how quality of instances can be judged.

We now have all concepts necessary to introduce the story concept. While instances are basically collections of facts, a *story* contains a subset of these facts, arranged in a certain order.

Definition 6.3.8 (Story) *Suppose $\mathcal{I}$ is a closed instance w.r.t. a generalized story schema $(\mathcal{E}, \mathcal{A}, \sim, \mathcal{G})$ and a source access table SAT, and $e \in \mathcal{E}$ is an entity. Then a* story *$\sigma(e, \mathcal{I})$ of size k, is a sequence of attribute-value pairs $\langle A_1, v_1\rangle, \ldots, \langle A_k, v_k\rangle$ such that for all $1 \leq i \leq k$, $A_i \in \mathcal{A}$ and $v_i = \mathcal{I}(e_i, A_i)$.*
A deconflicted story *w.r.t. a given conflict management policy is a sequence of attribute-value pairs $\langle A_1, v_1\rangle, \ldots, \langle A_k, v_k\rangle$ such that for all $1 \leq i \leq k$, $A_i \in \mathcal{A}$ and $v_i = \mathcal{I}^\sharp(e_i, A_i)$ where $\mathcal{I}^\sharp$ is a deconflicted instance w.r.t. χ.*

Note 6.3.2 *Throughout the rest of this chapter, I will use the word "story" to refer to both ordinary and deconflicted stories.*

Note that the above definition of a story only considers the content of a story and the order in which this content has to be presented. Actual presentation of a story to the audience will be discussed later in this chapter.

6.3.3 Optimal Stories

Not all stories are the same, even when they are on the same topic. For example, there are undoubtedly very boring books about Pentheus, just as there are illuminating and exciting books about him. While it is unwise to expect a computer to come up with an exceptional literary work, a user would still like to have a story that is cohesive, succinct and gets all the important facts across.

In Chapter 5 I have described the three CPR criteria [24] that affect the quality of video summaries: the *continuity*, the *priority*, and the *repetition*. As a story is essentially a kind of a summary, these three criteria apply to stories as well.

Let us see what each of the CPR criteria means when applied to a story. The *priority* determines the importance of facts included into a story. The *continuity* determines whether these facts are explained in the right logical order. Finally, the *repetition* determines the redundancy of included facts. Of course, in the last case we actually need the *non-repetition* rather than repetition.

As all three CPR criteria are subjective, let us define them in an abstract way. This will allow the story computation algorithms described later in this chapter to work with any notions of priority, continuity and non-repetition that fit within the abstract definitions given below.

Definition 6.3.9 (Story Evaluation Function) *Suppose $\mathcal{S}$ is the set of all possible stories about some entity e w.r.t. the same schema and source access table. Let χ, ϕ, and ρ be* arbitrary *functions from $\mathcal{S}$ to the set of real numbers $[0, 1]$ that measure the story's* continuity, priority, *and* repetition, *respectively. Given positive coefficients α, β, γ, these three functions can be aggregated into a single* story evaluation function

$$eval(s) = \alpha \cdot \chi(s) + \beta \cdot \phi(s) - \gamma \cdot \rho(s).$$

The evaluation function above consists of three weighted components. The first component, captured by the function ϕ measures the priority of the facts are included in the story. For example, the fact that Pentheus' mother was Agave is more important than the length of Pentheus' big toe. The second component, expressed through χ, describes how continuous the story is. Stories that deliver facts in a "conventional" order are more continuous than ones that jump wildly from one fact to another. The

third component concerns repetition – clearly, stories that repeat the same facts over and over again leave much to be desired.

Of course, there are multiple ways to compute the values of χ, ϕ, and ρ. Let us look at some possibilities.

Example 6.3.4 (Weight-Based Priority) *Assume the existence of a mapping* $pwt : \mathcal{E} \times \mathcal{A} \rightarrow [0,1]$ *that assigns the degree of importance to each entity-attribute pair. Then, one can compute the total priority of a story* $\{\langle A_1, v_1\rangle, \ldots, \langle A_k, v_k\rangle\}$ *as*

$$\phi(\{\langle A_1, v_1\rangle, \ldots, \langle A_k, v_k\rangle\}) = \frac{2}{k^2} \cdot \sum_{1 \leq i \leq k} (pwt(e, A_i) \cdot (k - i + 1)).$$

Example 6.3.5 (Knowledge-Based Priority) *Every attribute value can be an entity on its own and may therefore have some attributes attached to it. The following priority function will favor stories that tell the user about well described entities first:*

$$\phi(\{\langle A_1, v_1\rangle, \ldots, \langle A_k, v_k\rangle\}) = \frac{2 \cdot \sum_{1 \leq i \leq k}((k - i + 1) \cdot card(\{A \in \mathcal{A} \mid \mathcal{I}(v_i, A) \neq \perp\}))}{k^2 \cdot max_{e \in \mathcal{E}} card(\{A \in \mathcal{A} \mid \mathcal{I}(e, A) \neq \perp\})}.$$

As $\mathcal{I}$ *is a* partial function*, we use the notation* $\mathcal{I}(e, A) \neq \perp$ *above to denote that* $\mathcal{I}(e, A)$ *is defined.*

Example 6.3.6 (Reference-Based Priority) *An attribute value can be referred to by attributes of other entities irrespective of whether it is a generic value (such as "36" or "green") or an entity on its own (such as "Agave"). In the second case, it can be beneficial to give often referred entities better placement in a story. This can be achieved with the following priority function:*

$$\phi(\{\langle A_1, v_1\rangle, \ldots, \langle A_k, v_k\rangle\}) = \frac{2 \cdot \sum_{1 \leq i \leq k}((k - i + 1) \cdot card(\{\langle e, A\rangle \mid v_i \in \mathcal{E} \wedge \mathcal{I}(e, A) = v_i\}))}{k^2 \cdot max_{e' \in \mathcal{E}} card(\{\langle e, A\rangle \mid \mathcal{I}(e, A) = e'\})}.$$

Example 6.3.7 (Weight-Based Continuity) *Assume the existence of a mapping* $cwt : \mathcal{A} \times \mathcal{A} \rightarrow [0,1]$ *that expresses the degree of connectedness between each two attributes. Such a function can, for example, be specified via a table which shows for*

each pair of attributes, what their degree of connectedness is. Then, one can compute the total continuity of a story $\{\langle A_1, v_1\rangle, \ldots, \langle A_k, v_k\rangle\}$ *as*

$$\chi(\{\langle A_1, v_1\rangle, \ldots, \langle A_k, v_k\rangle\}) = \frac{2}{k} \cdot \sum_{1 \leq i < k} \sum_{i < j \leq k} \frac{cwt(A_i, A_j)}{|j - i|}.$$

We will additionally require that for any unique $A, A_1, A_2 \in \mathcal{A}$, $cwt(A, A) = 1$ *and* $cwt(A_1, A_2) \geq cwt(A_1, A) + cwt(A, A_2) - 1$.

Example 6.3.8 (Time-Based Continuity) *As attributes may have timevalues, we may want to arrange them in order of increasing time to achieve continuity. This can be done with the following conitnuity function:*

$$\chi(\{\langle A_1, v_1\rangle, \ldots, \langle A_k, v_k\rangle\}) = \frac{2}{k \cdot (k-1)} \cdot \sum_{1 \leq i < k} \sum_{i < j \leq k} \begin{cases} 0 & if\ v_i = (-, t_i, -) \wedge v_j = (-, t_j, -) \wedge t_i > t_j \\ 1 & otherwise \end{cases}.$$

Example 6.3.9 (Weight-Based Repetition) *Assume the existence of a mapping* $rwt : \mathcal{A} \times \mathcal{A} \rightarrow [0, 1]$ *that expresses the degree of similarity between each pair of attributes. Such a function can, for example, be specified via a table which shows for each pair of attributes, what their degree of similarity is. Then, one can compute the total repetition of a story* $\{\langle A_1, v_1\rangle, \ldots, \langle A_k, v_k\rangle\}$ *as*

$$\rho(\{\langle A_1, v_1\rangle, \ldots, \langle A_k, v_k\rangle\}) = \frac{2 \cdot \sum_{1 \leq i < k} \sum_{i < j \leq k} (rwt(A_i, A_j) \cdot card(v_i) \cdot card(v_j))}{k \cdot (k-1) \cdot max_{1 \leq i \leq k} card^2(v_i)}.$$

We will additionally require that for any unique $A, A_1, A_2 \in \mathcal{A}$, $rwt(A, A) = 1$, $rwt(A_1, A_2) = rwt(A_2, A_1)$, *and* $rwt(A_1, A_2) \geq rwt(A_1, A) + rwt(A, A_2) - 1$.

Armed with a collection of facts in the shape of an instance and a story evaluation function, we can now define what it means to create an *optimal story*.

Problem 1 (Optimal Story) *Given an instance* $\mathcal{I}$*, a positive integer* k*, and an entity* $e \in \mathcal{E}$ *as input, find a* story $\sigma(e, \mathcal{I})$ *of size* $\leq k$ *that maximizes the value of a given evaluation function. In this case,* $\sigma(e, \mathcal{I})$ *is called an* optimal story.

The next section will present algorithms to compute optimal (and suboptimal) stories.

6.4 Story Computation Algorithms

This section starts with the algorithms that help produce full, closed, and deconflicted instances, with respect to a given entity. I will then present a computationally complex algorithm that produces optimal stories and its faster suboptimal version. Finally, we will look at a few heuristic based algorithms which, while producing suboptimal results, may be more practical due to their lower computational complexity.

6.4.1 Building Instances

In order to create a story, one must first collect and generalize all known facts about story's subject and remove all possible conflicts between these facts. We have seen before in this chapter that such collections of facts are abstracted with *instances*. Let us then go over algorithms that, given an entity of interest, compute full, closed, and deconflicted instances with respect to this entity.

The first algorithm, called $FullI()$, takes an entity e and a source access table SAT as input and returns as output, the portion of the full instance generated by SAT which pertains to e.

```
Algorithm FullI(e,SAT)
    e is an entity
    SAT is a source access table
begin
    Result := ∅
    for each ⟨A, s, f_{A,s}⟩ in SAT do
        if f_{A,s}(e) ≠ ∅ then
            if exists ⟨A, V⟩ ∈ Result then
                Result := Result \ {⟨A, V⟩}
                Result := Result ∪ {⟨A, V ∪ {f_{A,s}(e)}⟩}
            else
                Result := Result ∪ {⟨A, {f_{A,s}(e)}⟩}
            end if
        end if
    end for
    return Result
end
```

It is easy to see that $FullI()$ executes in time proportional to the size of SAT. Given a full instance $\mathcal{I}$, an equivalence relation $\sim$, and a generalization mapping $\mathcal{G}$, on can now compute the *closed instance* with respect to an entity e by using the following $ClosedI()$ algorithm.

```
Algorithm ClosedI(e,I)
    e is an entity
    I is a full instance
begin
    Result := ∅
    for each ⟨A, V⟩ = I(e, A) do
        D := V
        while exists v ∈ V do
            S := {v' ∈ V | v' ~ v}
            D := D ∪ {Γ_A(S)}
            V := V \ S
        end while
        Result := Result ∪ {⟨A, D⟩}
    end for
    return Result
end
```

The $ClosedI()$ algorithm executes in time proportional to the number of attributes in its input multiplied by the average number of values per attribute. Suppose now that χ is a conflict management policy. The following $DeconfI()$ algorithm takes an entity of interest e and a closed instance $\mathcal{I}$ and applies χ to return the *deconflicted instance* with respect to e.

```
Algorithm DeconfI(e,I)
    e is an entity
    I is a closed instance
begin
    Result := ∅
    for each ⟨A, V⟩ = I(e, A) do
        Result := Result ∪ {⟨A, χ(V)⟩}
    end for
    return Result
end
```

The $DeconfI()$ algorithm executes in time proportional to the number of attributes in its input multiplied by the time taken by the conflict management policy χ.

We are now ready to create a stories.

6.4.2 Optimal Story Creation

Given an entity e and a source access table SAT, the $OptStory()$ algorithm will find an *optimal* story of length k by maximizing the value of the evaluation function.

```
Algorithm OptStory(e,SAT,k)
    e is an entity
    SAT is a source access table
    k is the requested story size
begin
    I := DeconfI(ClosedI(e, FullI(e, SAT)))
    return RecStory(∅, I, k)
end
```

The $OptStory()$ algorithm creates a deconflicted instance $\mathcal{I}$ with respect to e and calls the recursive $RecStory()$ algorithm. The $RecStory()$ enumerates over all possible stories of k or fewer attributes that can be derived from the given data and returns the best story with respect to the evaluation function $eval()$.

```
Algorithm RecStory(Story,I,k)
    Story is the story so far
    I is a partial instance
    k is the remaining story size
begin
    ⟨BestS, BestW⟩ := ⟨Story, eval(Story)⟩
    if k > 0 then
        for each ⟨A, v⟩ ∈ I do
            S := Story with ⟨A, v⟩ attached to the tail
            ⟨S, W⟩ := RecStory(S, I \ {⟨A, v⟩}, k − 1)
            if W > BestW then ⟨BestS, BestW⟩ := ⟨S, W⟩
        end for
    end if
    return ⟨BestS, BestW⟩
end
```

Given n attributes, the $RecStory()$ algorithm will have to sort through $\sum_{0 \leq i \leq k} \frac{n!}{(n-i)!}$ stories. Even if we restrict the algorithm to the k-length stories, it will still have to consider $\frac{n!}{(n-k)!}$ stories. To make story creation more manageable, let us consider the following algorithm, similar to the SEA algorithm from Chapter 5.

```
Algorithm RecStory+(Story,I,k,b)
    Story is the story so far
    I is a partial instance
    k is the remaining story size
    b is the branching factor
begin
    ⟨BestS, BestW⟩ := ⟨Story, eval(Story)⟩
    Q is a priority queue
    if k > 0 then
        for each ⟨A, v⟩ ∈ I do
            S := Story with ⟨A, v⟩ attached to the tail
            Q.add(S, eval(S))
            if length(Q) > b then Q.delete(tail(Q))
        end for
        for each SS ∈ Q do
            ⟨S, W⟩ := RecStory(SS, I \ SS, k − 1)
            if W > BestW then ⟨BestS, BestW⟩ := ⟨S, W⟩
        end for
    end if
    return ⟨BestS, BestW⟩
end
```

The $RecStory^{+}()$ algorithm, essentially limits search at each step to the b best stories w.r.t. the evaluation function. Given n attributes, this algorithm only considers $1 + \sum_{0 \leq i < k}(b^i \cdot (n - i))$ stories. We will call $OptStory^{+}()$ the algorithm that calls $RecStory^{+}()$.

6.4.3 Heuristic Based Algorithms

The $GenStory()$ algorithm given below is similar to the CPRgen algorithm from Chapter 5. It uses the genetic programming approach to generate suboptimal stories in a reasonable amount of time.

```
Algorithm GenStory(e,SAT,k,N,δ)
    e is an entity
    SAT is a source access table
    k is the requested story size
    N is the maximal number of iterations
    δ is the desired fitness threshold
begin
    I := DeconfI(ClosedI(e, FullI(e, SAT)))
    R := ⌈card(I) / k⌉
```

```
    Q := R stories of k attributes randomly chosen from I
    for j ∈ [1, N]
        for i ∈ [1, R]
            S := a solution randomly chosen among the ones in Q
            Choose random ⟨A, v⟩ ∈ I and ⟨A', v'⟩ ∈ S
            Replace ⟨A', v'⟩ in S with ⟨A, v⟩
            Q := Q ∪ {S}
            Q := Q \ {S'} where ∀S ∈ Q eval(S) ≥ eval(S')
            if max_{S1,S2∈Q} |eval(S1) − eval(S2)| ≤ δ then
                Return best solution from Q
            end if
        end for
    end for
    Return best solution from Q
end
```

The $GenStory()$ algorithm starts by obtaining a deconflicted instance $\mathcal{I}$ with respect to e. It then creates an initial population of stories Q and proceeds to replace a random attribute in a random story taken from Q with another random attribute. The resulting new story is then added to Q and the worst story (w.r.t. $eval()$) is deleted from Q. This process repeats until all stories in Q have approximately the same worth (w.r.t. the value of δ) or the maximal number of iterations N is reached.

The following $DynStory()$ algorithm is similar to the CPRdyn algorithm from Chapter 5. It uses the dynamic programming approach to generate suboptimal stories in a reasonable amount of time.

```
Algorithm DynStory(e,I,k)
    e is an entity
    I is the strong instance
    k is the requested story size
begin
    I := DeconfI(ClosedI(e, FullI(e, SAT)))
    S := solution of k attributes randomly chosen from I
    I := I \ S
    while I ≠ ∅
        subs := false
        r := 1
        while r < k and subs = false
            S' := S with ⟨A_r, v_r⟩ replaced with first(I)
            if eval(S) < eval(S') then
                S := S'
                Add ⟨A_r, v_r⟩ to the tail of I
```

```
            subs := true
        else
            r := r + 1
        end if
    end while
    remove first(I) from I
  end while
  return S
end
```

The $DynStory()$ algorithm also starts by obtaining the deconflicted instance $\mathcal{I}$ and creating a single random solution S. It then treats $\mathcal{I}$ as an *ordered list* of candidates (as opposed to a set) and tries to replace each attribute in S with the first attribute from this list. As soon as a better solution is found, it takes the place of S. The algorithm terminates when the list of candidates is exhausted.

6.5 Implementation

Our group has developed a scalable story-telling architecture and implemented the STORY prototype system for automatically extracting facts from heterogeneous data sources, creating stories based on these facts, and delivering these stories to client devices, such as computers, PDAs, cell phones, and so forth. Figure 6.3 describes the architecture of our system. It assumes that there are two classes of users:

- *Application developers* build story-telling applications such as the Pompeii tour guide or the military intelligence-on-request application. Application developers interact with the system using the *STORY Developer Interface* in Figure 6.3.

- *End users* use story-telling applications built by the application developers. End users request and view stories using the *STORY Client*, as shown in Figure 6.3.

Our system contains all necessary tools for both kinds of users.

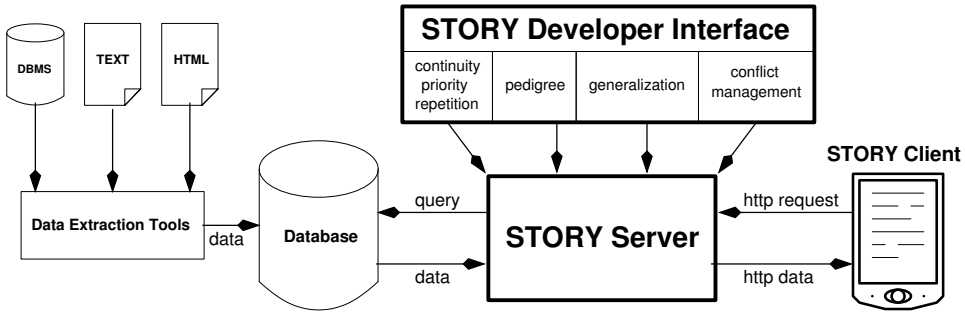


Figure 6.3: System Architecture.

The system consists of the Data Extraction Tools (written in C#) that help extracting facts about entities from the data sources; the STORY Server (written in C++ and running on an Apache-equipped Solaris machine) that builds stories and delivers them over HTTP protocol to both generic Web browsers and specialized clients (such as a custom PocketPC client used in the Pompeii tour guide); and STORY Developer Interface (again in C#) that gives application developer control over data extraction and story creation process. When creating a story-telling application, the STORY application developer typically goes through the following steps:

STEP 1: Extraction of attribute values. The developer needs to specify methods used to extract attribute values from heterogenous data sources, such as text, HTML documents, relational databases, and so forth. I will describe existing extraction methods in detail later in this section.

STEP 2: Description of continuity, priority, and repetition. The developer specifies the kinds of continuity, priority, and repetition functions to be used. He can either use ones mentioned earlier in this chapter, or add new ones that better fit his needs. For example, attributes can be grouped into hierarchies and priorities set for entire hierarchical branches, as well as for individual attributes.

STEP 3: Description of the conflict management. The developer specifies what kind of conflict management function he wants to use. He can either select from a few predefined functions, or add his own function.

STEP 4: Selection of the story creation algorithm. The developer describes which of the several story creation algorithms should be used. We currently offer a choice of $OptStory^{+}()$, $GenStory()$, or $DynStory()$ algorithms.

STEP 5: Description of the story output. The developer specifies the format in which stories will be presented to users. Using custom client applications, a story can be presented in a wide variety of different ways. For example, our client implementationa can present it as a table or a tree of attributes, or a narrative in English, Spanish, or Italian language. The narrative can be made of sentences taken from the data sources (in the language of these sources), or generated using templates. In the latter case, its output language can be changed with a relative ease, but the application developer has to supply a template for each attribute. Such a template is a plain text sentence with special tags "%e", "%a", and "%v" specifying how to express the fact that "entity e's attribute a has a value v".

STEP 6: Story prefetching. The developer can supply a map with spatial locations of all or some entities and specify methods to prefetch stories to user's device (e.g. a PDA or the like). In some applications, such as the Pompeii tour guide, there is an advantage in tracking users' locations and prefetching stories based on their proximity to certain objects of interest.

6.6 Extracting Attribute Values

The attribute value extraction is perhaps the most difficult problem we have faced when implementing the STORY system. The present implementation extracts data from the Web (with some minimal human assistance). With the algorithms presented in the rest of this section, the data can also be extracted from relational tables and XML files, although the lack of datasets relevant to our sample applications has so far discouraged us from actively using these two sources.

6.6.1 Relational Sources

Extracting attribute values from relational sources is relatively easy, as the data is already well organized. Consider a relational table $T = \{c_1, \ldots, c_m, \ldots, c_n\}$ where $c_1, \ldots, c_n$ are columns and $c_1, \ldots, c_m$ are also keys. It is logical to assume that each column c_j corresponds to an attribute with the same name and contains values for this attribute. Then each row in T corresponds to a set of attributes. As a row is addressed by its key column values, one can assume that its corresponding entity can be stored in any of the key fields. Then for each two columns $c_{1\leq i\leq m}, c_{1\leq j\leq n}$ we add the following entry to the SAT table:

$$\langle c_j,\ T : c_i,\ f_{c_j,T:c_i}(e) = \pi_{c_j}\sigma_{c_i=e}T\rangle.$$

In other words, given a table T as the source, we obtain an attribute c_j for an entity e by looking for all table rows that can be referred by e and picking their c_j values.

6.6.2 XML Sources

Data extraction from XML documents is somewhat more complicated than from the relational tables, as XML has a flexible hierarchical data structure. Consider an XML

node $N = \langle name, value, \{c_1, \ldots, c_n\}\rangle$ where $c_1, \ldots, c_n$ are children nodes. Assuming that N is a root node in an XML document, and nodes may act both as entities and attributes, one can write the following algorithm to return a given attribute A for a given entity e.

```
Algorithm GetXMLAttr(N,e,A)
    N is the root XML node
    e is the entity
    A is the attribute
begin
    Result := ∅
    if N.value = e or N.name = e then
        for each child c of N such that c.name = A do
            Result := Result ∪ {c.value}
        end for
    else
        for each child c of N do
            Result := Result ∪ GetXMLAttr(c, e, A)
        end for
    end if
    return Result
end
```

The $GetXMLAttr()$ recursively finds all occurences of an entity e in the XML tree, collects all values for the requested attribute A, and returns the set of collected values. Notice that the algorithm tries to match e to both *node value* and *node name*. We can now enumerate all node names and values occuring in the XML tree as $A_1, \ldots, A_m$, and for each A_i, add a SAT table entry $\langle A_i, N, GetXMLAttr(N, e, A_i)\rangle$.

6.6.3 Web Sources

The process of extracting data from Web documents is a complex task. The current STORY implementation accomplishes it by looking for relevant documents using a common Web search engine (Google), performing semantic analysis of found documents, and extracting facts with the help of logical rules. The results are then presented to the application developer, who removes any mistakes introduced by the

extraction engine and uploads the data to the server. Following is a more detailed explanation of our data extraction process, step by step:

STEP 1: Document search. The extraction engine uses Google to search the Web for documents related to the entity of interest (e.g. "Pentheus") in the domain of interest (e.g. "Greek Mythology"). Returned documents are passed through an HTML parser that extracts from them significant pieces of text, while discarding the HTML formatting structure.

STEP 2: Lexical analysis. The extraction engine uses the publicly available WordNet lexical database to analyze the text and tags each word with its corresponding part of speech.

STEP 3: Name detection. The extraction engine uses some heuristics to recognize and classify named entities (such as personal, organizational, corporate, and geographic names and trademarks). The heuristic algorithm used in this step can be trained on a large corpora of data to improve its performance.

STEP 4: Disambiguation. The extraction engine uses common *pronoun resolution* and *word sense disambiguation* algorithms to produce an unambiguous version of the original text.

STEP 5: Semantic parsing. Finally, the rewritten text is analyzed by a semantic parser which applies a set of *semantic rules* to deduce the entity-attribute-value triples. Semantic rules are of the form *head* $\rightarrow$ *tail*, where *head* is a pattern to be matched to a sentence taken from the text. If the pattern matches, the *tail* says how to extract one or more entity-attribute-value triples from this pattern. The system will try to determine the relevant time interval for each triple and

tag it with the pedigree information, such as the source and the date when the document has been last updated. This data can be later used for conflict management, or to give users information on where each fact comes from.

The semantic rules can be either entered into the system manually by the application developer or learned semi-automatically from examples.

STEP 6: Data cleanup. The application developer is presented with the entity-attribute-value triples extracted by the engine. His job is to remove any erroneous triples that sometimes appear in the extraction results. When developer is satisfied with the quality of results, he submits the data for upload to the STORY server.

6.7 Conclusion

In this chapter, I have laid out a theoretical foundation for automated story-telling based on the data culled from heterogenous data sources (RDBMS, XML, Web). I have shown how story-worthy facts are collected, processed, arranged into stories, and how resulting stories can be rated with respect to continuity, priority, and non-repetition criteria. Furthermore, I have presented several algorithms to create stories. Finally, the chapter describes our prototype STORY system, its architecture and the process for creating story-telling applications.

The STORY project is an ongoing work. There is still a lot of space for improvement in our data extraction subsystem, in the template-based output mechanism, and the way system presents stories to users. The goal of the project is to make the system intelligent enough to perform data extraction on the fly, without human assistance, and create narrative that is both informative *and* engaging.

Chapter 7

Conclusion

Historically, computers were built to manipulate numbers and, later, text documents. There is a vast body of research related to these two kinds of data, with the relational algebra providing the common framework for organizing data.

The current abundance of cheap data storage, computing power, and rich presentation capabilities has made it possible to store and process large amounts of images, sound, and video. As result computers and computer networks gradually replace radio, television sets, tape and disk players as the primary source of audiovisual information and entertainment in today's homes.

The popularity of the multimedia data has stemmed research in multimedia storage, searching, and delivery in the past years. Unfortunately, while many specific properties of different media types have been addressed, there still isn't a common framework for organizing multimedia data in the same way relational algebra organizes numeric and text data. As of now, the most common method to "befriend" multimedia and traditional databases is to store multimedia files as "blobs" inside relational fields.

In this thesis, I have tried to fill the gap by proposing several relational-like al-

gebras for operating on PowerPoint, audio, and video data, which are kinds of multimedia most commonly used today. These algebras are devised in a way that addresses different representations of each media type, while leveraging their common properties. For example, the ADA audio database algebra represents audio with a set of time-ordered data sequences, including traditional waveforms, frequency spectrum changes, musical scores, or text transcripts. The VDA video database algebra represents video with a series of segments (blocks) characterized by features. While these features may come from different sources and represent different things (objects, events, paths, color distribution, movement), they all share such common properties as location at the screen and the percentage of this location occupied by the feature. The relative simplicity of the proposed algebras allows to prove some useful equivalences in them. Coupled with the cost model, these equivalences become the basis for the query optimization, as has been shown in this thesis. I have also presented some ways to accelerate query execution by indexing data and close examination of queries.

In addition to the algebraic approach to multimedia processing presented in the first chapters of this thesis, in the last chapters I present an alternative logical approach that uses logical reasoning to create short summaries of large multimedia files (Chapter 5) or facts obtained from heterogenous data sources (Chapter 6). To estimate the quality of produced summaries, I introduce the three criteria: (i) priority of the summarized information, (ii) continuity of a summary, (iii) amount of repetition in a summary. A summary is then composed of the available information by an algorithm maximizing the first two parameters (priority and continuity), while minimizing the third (repetition). I show several such summary composition algorithms.

It is my hope that concepts presented in this thesis will help other people to design

and implement large multimedia database systems able to interoperate with each other and produce “intelligent” answers to user queries.

Bibliography

[1] S. Adali, P. Bonatti, M.L. Sapino, and V.S. Subrahmanian. *A Multi-Similarity Algebra.* Proc. of ACM SIGMOD Conference on Management of Data, 1998, pp. 402-413.

[2] S. Adali, K.S. Candan, S.S. Chen, K. Erol, and V.S.Subrahmanian. *Advanced Video Information Systems.* ACM Multimedia Systems Journal, Vol. 4, 1996, pp. 172-186.

[3] S. Adali, M.L. Sapino, and V.S.Subrahmanian. *Interactive Multimedia Presentation Databases, I: Algebra and Query Equivalences.* ACM/Springer Multimedia Systems Journal, Vol. 8, Nr. 3, 2000, pp. 212-230.

[4] M. Albanese, M. Fayzullin, A. Picariello, and V.S. Subrahmanian. *The Priority Curve Algorithm for Video Summarization.* Proc. of 2nd ACM Intl. Workshop in Multimedia Databases, Arlington, Virginia, 2004.

[5] C. Baral, G. Gonzalez, and T.C. Som. *Design and Implementation of Display Specifications for Multimedia Answers.* Intl. Conference on Data Engineering, Orlando, Florida, 1998, pp. 558-565.

[6] R. Barzilay and M. Elhadad. *Using Lexical Chains for Text Summarization.* Proc. Intelligent Scalable Text Summarization Workshop, Madrid, Spain, 1997.

[7] T. Blum, D. Keislar, J. Wheaton and E. Wold. *Audio Databases with Content-Based Retrieval.* Proc. IJCAI Workshop on Intelligent Multimedia Information Retrieval, Montreal, Canada, 1995.

[8] T. Blum, D. Keislar, J. Wheaton and E. Wold. *Content-Based Classification, Search, and Retrieval of Audio.* IEEE Multimedia Magazine, Vol. 3(3), 1996, pp. 27-36.

[9] C. Bohm, S. Berchtold, and D.A. Keim. *Searching in High-Dimensional Spaces: Index Structures for Improving The Performance of Multimedia Databases* ACM Computing Surveys, Vol. 33(3), 2001, pp. 322-373.

[10] R.S. Boyer and J.S. Moore. *A Fast String Searching Algorithm.* Communications of the ACM, Vol. 20, 1977, pp. 762-772.

[11] M.C. Buchanan and P.T. Zellweger. *Automatically Generating Consistent Schedules for Multimedia Documents.* ACM/Springer Multimedia Systems Journal, Vol. 1(2), 1993.

[12] K. Chakrabarti, E. J. Keogh, S. Mehrotra, and M. J. Pazzani. *Locally Adaptive Dimensionality Reduction for Indexing Large Time Series Databases.* ACM Transactions on Database Systems, Vol. 27(2), 2002, pp. 188-228.

[13] K. Chan and W. Fu. *Efficient Time Series Matching by Wavelets.* Intl. Conference on Data Engineering, Sydney, 1999, pp. 126-133.

[14] S. Chawathe, H. Garcia-Molina, J. Hammer, Y. Papakonstantinou, J. Ullman, and J. Widom. *The Tsimmis Project: Integration of Heterogeneous Information Sources.* Proc. 100th Anniversary Meeting of the Information Processing Society of Japan, Tokyo, Japan, 1994, pp. 7-18.

[15] S. Chang, J.R. Smith, M. Beigi, and A. Benitez. *Visual Information Retrieval from Large Distributed Online Repositories.* Communications of the ACM, December 1997, pp. 63-71.

[16] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *"Introduction to Algorithms, 2nd Edition"* MIT Press, 2001.

[17] Y.F. Day, S. Dagtas, M. Iino, and A. Ghafoor. *Object-Oriented Conceptual Modeling of Video Data.* Proc. 11th International Conference on Data Engineering, Taipei, Taiwan, 1995, pp. 401-408.

[18] D. DeMenthon, D.S. Doermann, and V. Kobla. *Video Summarization by Curve Simplification.* Proc. ACM - Multimedia, Bristol, England, 1998, pp. 211-218.

[19] Y.F. Day, A. Khokhar, S. Dagtas, and A. Ghafoor. *A Multi-Level Abstraction and Modeling in Video Databases.* ACM/Springer Multimedia Systems Journal, Vol. 7(5), 1999, pp. 409-423.

[20] A. Duda, R. Weiss, and D.K. Gifford. *Content-Based Access to Algebraic Video.* 1994.

[21] C. Faloutsos. *"Searching Multimedia Databases by Content"* Kluwer, 1996.

[22] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. *Fast Subsequence Matching in Time-Series Databases.* ACM SIGMOD Intl. Conference, Minneapolis, Minnesota, 1994, pp. 419-429.

[23] M. Farach. *Optimal Suffix Tree Construction with Large Alphabets.* Proc. 38th Annual Symposium on the Foundations of Computer Science, New York, 1997.

[24] M. Fayzullin, A. Picariello, M.L. Sapino, and V.S. Subrahmanian. *The CPR Model for Summarizing Video.* Proc. of 1st ACM Intl. Workshop in Multimedia Databases, New Orleans, Louisiana, 2003, pp. 2-9.

[25] M. Fayzullin and V.S. Subrahmanian. *An Algebra for PowerPoint Sources.* Multimedia Tools and Applications, Vol. 2(3), 2004, pp. 273-301.

[26] M. Fayzullin and V.S. Subrahmanian. *Optimizing Selection and Mixing in Audio Databases.* Workshop on Multimedia Information Systems, College Park, Maryland, 2004.

[27] J.T. Foote. *Content-Based Retrieval of Music and Audio.* Multimedia Storage and Archiving Systems II, Proc. of SPIE, Vol. 3229, 1997, pp. 138-147.

[28] J. Foote and S. Uchihashi. *Summarizing Video Using a Shot Importance Measure and a Frame-Packing Algorithm.* Proc. of the Int. Conf. on Acoustics, Speech, and Signal Processing, Phoenix, 1999, Vol. 6, pp. 3041-3044.

[29] M. Furini and D. Towsley. *Real-Time Traffic Transmission Over the Internet.* IEEE Transactions on Multimedia, Vol. 3(1), 2001, pp. 33-40.

[30] L. Gao, J. Kurose, and D. Towsley. *Efficient Schemes for Broadcasting Popular Videos.* ACM/Springer Multimedia Systems Journal, Vol. 8(4), 2002, pp. 284-294.

[31] L. Gao and D. Towsley. *Threshold-Based Multicast for Continuous Media Delivery.* IEEE Transactions on Multimedia, Vol. 3(4), 2001, pp. 405-414.

[32] U. Gargi, R. Kasturi, and S.H. Strayer. *Performance Characterization of Video-Shot Change Detection Methods.* IEEE Trans. on Circuits Systems Video Technology, Vol. 10(1), 2000, pp. 1-13.

[33] A. Ghias, J. Logan, D. Chamberlin and B. Smith. *Query by Humming: Musical Information Retrieval in an Audio Database.* ACM Multimedia, 1995, pp. 231-236.

[34] Y. Gong and X. Liu. *Video Summarization Using Singular Value Decomposition.* Proc. of Computer Vision and Pattern Recognition, 2000, pp. 174-180.

[35] R. C. Gonzales and P. Winz. *"Digital Image Processing"* Addison-Wesley Publishing Company, Knoxville, Tennesee, 1987.

[36] B. Gunsel and A.M. Tekalp. *Content-Based Video Abstraction.* IEEE Proc. Int'l Conf. on Image Processing, Chicago, Illinois, 1998.

[37] V. Hakkoymaz, J. Kraft, and G. Ozsoyoglu. *Constraint Based Automation of Multimedia Presentation Assembly.* ACM/Springer Multimedia Systems Journal, Vol. 7(6), 1999.

[38] A. Hanjalic. *Shot-Boundary Detection: Unraveled and Resolved?* IEEE Trans. Circuits Systems Video Technology, Vol. 12, 2002, pp. 90-105.

[39] L. He, E. Sanocki, A. Gupta and J. Grudin. *Auto-Summarization of Audio-Video Presentations.* ACM Proc. on Multimedia, 1999, pp. 489-498.

[40] E. Hovy and C.Y. Lin. *Automated Text Summarization in SUMMARIST.* In Mani and Maybury, 1998, pp. 18-24.

[41] J.L. Hsu, A.L.P. Chen, H.C. Chen, N.H. Liu. *The Effectiveness Study of Various Music Information Retrieval Approaches.* Proc. ACM Int. Conf. on Information and Knowledge Management, McLean, 2002, pp. 422-429.

[42] M.Y. Kim and J. Song. *Multimedia Documents with Elastic Time.* ACM Multimedia Conference, 1995.

[43] F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas. *Fast Nearest Neighbor Search in Medical Image Databases.* Proc. 22nd International Conference on Very Large Data Bases, Mumbai, India, 1996, pp. 215-226.

[44] T. Kanade, M. Smith, S. Stevens, and H. Wactlar. *Intelligent Access to Digital Video: The Informedia Project.* IEEE Computer, Vol. 29(5), 1996, pp. 46-52.

[45] J. Korst and V. Pronk. *Storing Continuous Media Data on a Compact Disc.* ACM/Springer Multimedia Systems Journal, Vol. 4, 1996, pp. 187-196.

[46] T. Lee, L. Sheng, T. Bozkaya, N.H. Balkir, Z.M. Ozsoyoglu, and G. Ozsoyoglu. *Querying Multimedia Presentations Based on Content.* IEEE Trans. on Knowledge and Data Engineering, Vol. 11(3), 1999.

[47] K. Lemström, A. Haapaniemi and E. Ukkonen. *Retrieving Music - To Index or not to Index.* ACM Multimedia, September 1998, Bristol, UK.

[48] D. Li and H. Lu. *Model Based Video Segmentation.* IEEE Trans. Circuits Systems Video Technology, Vol. 5, 1995, pp. 533-544.

[49] J. Lin and Z.M. Ozsoyoglu. (1997) *Processing OODB Queries by O-Algebra.* Proc. 8th International Conference on Information and Knowledge Management, Rockville, Maryland, 1996, pp. 134-142.

[50] R. Lienhart, S. Pfeiffer, and W. Effelsberg. *The MoCA Workbench: Support for Creativity in Movie Content Analysis.* Proc. IEEE Conf. on Multimedia Computing and Systems, Hiroshima, Japan, 1995, pp. 314-321.

[51] T.D.C. Little and A. Ghafoor. *Synchronization and Storage Models for Multimedia Objects.* IEEE J. on Selected Areas of Communications, Vol. 8(3), 1990, pp. 413-427.

[52] T.D.C. Little and A. Ghafoor. *Interval-Based Conceptual Models for Time-Dependent Multimedia Data.* IEEE TKDE (Multimedia Information Systems), Vol. 5(4), 1993, pp. 551-563.

[53] W.Y. Ma and B.S. Manjunath. *NETRA: A Toolbox For Navigating Large Image Databases.* IEEE Multimedia Systems Journal, Vol. 7(3), 1999, pp. 184-198.

[54] D. Marcu. *From Discourse Structures to Text Summaries.* Proc. Intelligent Scalable Text Summarization Workshop, Madrid, Spain, 1997, pp. 82-88.

[55] H.Martin and R.Lozano. *Dynamic Generation of Video Abstracts Using an Object Oriented Video DBMS.* Networking and Information Systems Journal, Vol. 3(1), 2000, pp. 53-75.

[56] R.J. McNab, L.A. Smith, D. Bainbridge and I.H. Witten. *The New Zealand Digital Library MELody inDEX.* Digital Libraries Magazine, May 1997.

[57] S.B. Moon, J. Kurose, and D. Towsley. *Packet Audio Playout Delay Adjustment: Performance Bounds and Algorithms.* ACM/Springer Multimedia Systems Journal, Vol. 6(1), 1998, pp. 17-28.

[58] H.R. Naphide and T.S. Huang. *A Probabilistic Framework for Semantic Video Indexing, Filtering, and Retrieval.* IEEE Transactions on Multimedia, Vol. 3(1), 2001, pp. 141-151.

[59] E. Oomoto and K. Tanaka. *OVID: Design and Implementation of a Video-Object Database System.* IEEE TKDE (Multimedia Information Systems), Vol. 5(4), 1993, pp. 629-643.

[60] I.B. Ozer and W.H. Wolf. *A Hierarchical Human Detection System in (Un)Compressed Domains.* IEEE Transactions on Multimedia, Vol. 4(2), 2002, pp. 283-300.

[61] M.T. Ozsu, D. Szafron, G. El-Medani, and C. Vittal. *An Object-Oriented Multimedia Database System for a News-on-Demand Application.* ACM/Springer Multimedia Systems Journal, Vol. 3(5-6), 1995, pp. 182-203.

[62] S.C. Pei and Y.Z. Chou. *Efficient MPEG Compressed Video Analysis Using Macroblock Type Information.* IEEE Transactions on Multimedia, Vol. 1(4), 1999, pp. 321-333.

[63] E. Petrakis and C. Faloutsos. *Similarity Searching in Medical Image Databases.* IEEE TKDE, Vol. 9(3), 1997, pp. 435-447.

[64] A. Picariello, M.L. Sapino, and V.S. Subrahmanian. *A Video Database Algebra.* 2003

[65] E. Pollastri and G. Haus. *An Audio Front End for Query-by-Humming Systems.* Music Information Retrieval, 2001.

[66] I. Popivanov and R. J. Miller. *Similarity Search Over Time Series Data Using Wavelets.* Intl. Conference on Data Engineering, San Jose, California, 2002, pp. 802-813.

[67] D. Rafiei. *On Similarity-Based Queries for Time Series Data.* Intl. Conference on Data Engineering, Sydney, Australia, 1999.

[68] T.J. Rogers, R. Ross, and V.S. Subrahmanian. *IMPACT: A System for Building Agent Applications.* Journal of Intelligent Information Systems, Vol. 14(2-3), 2000, pp. 95-113.

[69] P. Salosaari and K. Jarvelin. *MUSIR - A Retrieval Model for Music.* Technical Report RN-1998-1, University of Tampere, Department of Information Studies, July 1998.

[70] H. Samet. *"The Design and Analysis of Spatial Data Structures"* Addison-Wesley, 1990.

[71] P. Schauble. *"Content-Based Information Retrieval from Large Text and Audio Databases"* Kluwer, 1997.

[72] Y. Shibata, T. Matsumoto, M. Takeda, A. Shinohara and S. Arikawa. *A Boyer-Moore Type Algorithm for Compressed Pattern Matching.* Proc. 11th Annual Symposium on Combinatorial Pattern Matching, Springer Verlag, 2000, pp. 181-194.

[73] A. Silberschatz, H. Korth, and S. Sudarshan. *"Database System Concepts"* McGraw-Hill, 1999.

[74] A. Soffer. *Image Categorization Using Texture Features.* Proc. 4th International Conference on Document Analysis and Recognition, Ulm, Germany, 1997, pp. 233-237.

[75] M.A. Stricker and M. Orengo. *Similarity of Color Images.* SPIE Proceedings, Vol. 2420, 1995, pp. 381-392.

[76] V.S. Subrahmanian. *"Principles of Multimedia Database Systems"* Morgan Kaufmann, 1998.

[77] V.S. Subrahmanian, S. Adali, A. Brink, R. Emery, J.J. Lu, A. Rajput, T.J. Rogers, R. Ross, and C. Ward. *HERMES: Heterogeneous Reasoning and Mediator System.*

[78] M.J. Swain and D.H. Ballard. *Color Indexing.* Int. Journal of Computer Vision, Vol. 7(1), 1991, pp. 11-32.

[79] B.T. Truong, C. Dorai, and S. Venkatesk. *New Enhancements to Cut, Fade, and Dissolve Detection Processing Video Segmentation.* ACM Multimedia, 2000, pp. 219-227.

[80] J.D. Ullman. *"Principles of Database Systems, 2nd Edition"* W.H. Freeman & Co, 1982.

[81] N. Vasconcelos and A. Lippman. *Bayesian Modeling of Video Editing and Structure: Semantic Features for Video Summarization and Browsing.* IEEE Proc. Int'l Conf. on Image Processing, Chicago, Illinois, 1996, pp. 153-157.

[82] J.Z. Wang, G. Wiederhold, O. Firschein, and S.X. Wei. *Content-Based Image Indexing and Searching Using Daubechies' Wavelets.* International Journal of Digital Libraries, Vol. 1(4), 1998, pp. 311-328.

[83] P. Weiner. *Linear Pattern Matching Algorithms.* Proc. 14th IEEE Annual Symp. on Switching and Automata Theory, 1973, pp. 1-11.

[84] D.A. White and R. Jain. *Similarity Indexing: Algorithms and Performance.* SPIE Proceedings, Vol. 2670, San Diego, 1996, pp. 62-73.

[85] M.J. Witbrock and A.G. Hauptmann. *Speech Recognition in a Digital Video Library.* J. of the American Society for Information Science (JASIS), Vol. 49(7), 1998, pp. 619-632.

[86] I. Yahiaoui, B. Merialdo, and B. Huet. *Generating Summaries of Multi-Episode Video.* IEEE Int. Conf. on Multimedia and Expo, 2001, pp. 22-25.

[87] B. Yi and C. Faloutsos. *Fast Time Sequence Indexing for Arbitrary LP-Norms.* VLDB Intl. Conference, Cairo, Egypt, 2000, pp. 385-394.