DATA MANAGEMENT OF MUSICAL INFORMATION

by

W. B. Rubenstein

Memorandum No. UCB/ERL M87/69

8 June 1987

DATA MANAGEMENT OF MUSICAL INFORMATION

by

William Bradley Rubenstein

Memorandum No. UCB/ERL M87/69

8 June 1987

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Data Management of Musical Information

William Bradley Rubenstein

Extended Abstract

This dissertation explores various issues related to the application of computer data management techniques to musical information. The contribution of this work is twofold:

- It extends an existing data model (the entity-relationship model) to support a database schema for musical information.
- It develops particular data management access methods to effectively manipulate information in the musical database.

Because the music domain has not previously been considered as an application area for database management systems, this thesis begins with a discussion of existing research regarding data management in a variety of "unusual" data domains. Also, the current role of computer technology in music applications is briefly surveyed. These discussions provide the context for the presentations which follow. Signal processing applications used for music synthesis are not included in this research, as they have been considered extensively elsewhere.

A close look at existing computer systems that manipulate musical information allows us to determine the types of musical information that should be supported by a music database. For the purposes of this research, musical information includes sound, graphical representations (such as musical scores), bibliographic information, and conceptual representations of music (such as structural descriptions of musical compositions). These various types of musical information are analyzed, focusing particularly on the conceptual representations necessary to formally encode musical scores expressed in *Common Musical Notation* (CMN). The entity-relationship model is taken as a starting point for a CMN database schema.

A new feature is added to the entity-relationship model to represent the notion of ordered sets of entities. Typically, an "ordering" occurs when one entity *consists of* an ordered set of other entities. This property is called *hierarchical ordering*. Such a relationship occurs, for example, when an ordered set of notes constitutes a chord, or an ordered set of measures forms a movement of a composition.

An *inherited attribute* is an attribute of an entity whose value is a function of attribute values in other related entities. A method is proposed for representing attribute inheritance among entities, and various approaches to the problem of managing this inheritance among entities within the music database are considered. Several examples demonstrate that inheritance under hierarchical ordering is more complex than that supported by standard generalization hierarchies. It is demonstrated how a relational view mechanism may be used to implement this attribute inheritance.

Using these two data modeling tools, hierarchical ordering and attribute inheritance, a schema is developed for CMN. Entities from the CMN schema are divided into groups according to various aspects of the information: temporal, timbral, and graphical. The schema is built up from the interrelationships among entities within each group.

To implement hierarchical ordering using existing relational database technology, extensions to relational access methods are developed. Entity ordering is supported by the introduction of *ordered relations.*

One form of attribute inheritance found in our application involves an attribute whose value depends on an "aggregate function" computed over its related entities. Aggregate functions common in database systems include count, sum, average, minimum, and maximum. A data access method is presented which supports this special case much more efficiently than the relational view mechanism. The data structure underlying this method, known as the A-tree, provides a general solution for the support of:

- user-defined aggregate functions, and
- user-defined orderings, including hierarchical orderings.

The performance behavior for A-trees is shown to be similar to that of the well-known B-tree structure on which it is based.

As an example of the use of ordered relations and A-trees, the manipulation of time and events in the musical database is explored in detail. A model of temporal information is presented based on *time lines* and hierarchically ordered *event sets*.

The dissertation closes with a summary of questions left open by the present research, particularly those remaining issues that need be addressed in order to develop a functional database system appropriate to the management of musical information.

Acknowledgements

A number of people deserve my thanks for their help in the preparation of this dissertation.

Foremost, I would like to thank Professor Michael Stonebraker, my research advisor. He has been a constant source of good advice and encouragement. To my knowledge, there has not previously been an inter-disciplinary thesis between music and computer science in our department, and the lack of precedent made this thesis somewhat more of a risk for him to sponsor. What we have learned in the mean time has I hope made the risk worthwhile. In any case, I owe Professor Stonebraker my deepest gratitude for his support, both material, intellectual and, for lack of a better word, spiritual. His enthusiasm has been unflagging.

The process of generating a dissertation which covers such widely divergent fields requires patience and forbearance on the part of its readers. I have been especially fortunate to have Professors Richard Karp, in the computer science department, and Richard Felciano, in the music department, in this regard.

I would like to thank the members of the INGRES group with whom many fruitful conversations were held over both the form and the content of this thesis, especially Eric Hanson, onto whose desk my papers continually overflowed. Also Margaret Butler, Yannis Ioannidis, and Timos Sellis, into whose office *I* occasionally overflowed. These people supplied both technical competence and moral support during this research.

Willis Johnson and Amy Jo Bilson helped edit the final versions of the dissertation. I would like to thank them also.

This research was supported by the National Science Foundation, grant #DMC-8504633, the Air Force Office of Scientific Research, grant #83-0254, and a fellowship from the Shell Oil Foundation.

Finally, I wish to thank the folks at the Coffee Connection for the 239 cups of hot chocolate required for the completion of this thesis.

Table of Contents

List of Figures

CHAPTER 1

Introduction

1.1. Introduction

Several research projects have recently focused on extending the applicability and usefulness of information management techniques and database systems to a variety of application areas. In the technical field, these include design data such as is generated by VLSI (Very Large Scale Integration) chip development and other CAD (Computer-Aided Design) processes [SRG83]. In the field of artificial intelligence, databases are being applied to the management of "knowledge bases" to support deduction and inference [DeF84]. In each of these cases, the data model, which serves as the primary tool for describing the representation of the data, has undergone successive extension and refinement. The entity-relationship model [Che76] serves as the starting point for our representation of musical information. This dissertation is concerned with those extensions and refinements necessary to support applications that manage this information.

Our formal definition of "musical information" begins in the next chapter. For the moment, it is worthwhile to note certain features of music that motivate our research into musical information as an interesting data management domain.

- Musical representations, such as music notation, have complex, rich semantics.

In particular, they must convey more information than simple lists, tables, and spatial representations, that are the mainstay of "traditional" database applications.

- The complexity of musical information is easily bounded, and therefore amenable to data management.

For example, cases of ambiguity which abound in natural language are more rare (though not unknown) in music. Additionally, the syntax and semantics of representations such as common musical notation are already reasonably well defined.

- The uses of musical information are, in a sense, limited and well understood.

For our purposes, typical examples of operations on musical data are production (e.g. composition and synthesis), editing, performance, and analysis. Intentionality and planning, which complicate artificial intelligence problems, are considered to be outside the domain of this research.

1.1.1. Organization of the Dissertation

The remainder of this dissertation is organized as follows. The next section, Section 1.2, introduces the Music Data Manager, its purpose as a system, and its potential clients. This sets the context for subsequent discussion regarding the operation of such a data manager. Section 1.3 concludes this chapter with an overview of research in related areas. The previous work that supports this thesis falls under several disparate research domains. Rather than presenting them all in detail here, discussion of research in each particular topic area will be pursued within the chapter devoted to that topic.

Chapter 2 discusses the various properties of musical information, and how they may be represented. It presents a wide variety of types of music information, including sound, graphical data, and bibliographic information, as well as conceptual representations of music.

Chapter 3 then discusses a semantic feature which pervades conceptual representations of music, *hierarchical ordering*. This construct has not been supported by existing data models. The chapter goes on to discuss the representation of inheritance in the music database. A class of inheritance induced by hierarchical ordering, designated *complex attribute inheritance*, is explored. It is shown that the relationships presented by the hierarchical ordering schema induce atypical inheritance semantics, similar to certain types of inheritance explored in the artificial intelligence domain, but quite different from that seen in database research.

Using the tools for representing hierarchical ordering developed in the previous chapter, chapter 4 explores the development of a database schema for *common musical notation* (CMN). The large number of entities are considered from the point of view of their temporal, timbral, and graphical aspects. For each such perspective, the hierarchical ordering relationships among the entities of CMN are discussed in detail.

Chapter 5 considers those issues related to the implementation of ordering and inheritance within a relational database system. The notion of *ordered relations* is introduced as a means of implementing hierarchical ordering. The remainder of chapter 5 is concerned with inheritance involving aggregate functions over ordered relations. By extending previous research regarding user-defined aggregate functions and ordered relations, an efficient access method is developed that supports an important class of inheritance functions in a general way. This access method is based on the A-tree data structure.

As an example of a hierarchical ordering application, chapter 6 addresses a very important aspect of the music information manager, that of time and events. A survey is presented of recent research that has considered the incorporation of time and temporal information into databases. Notions of *time lines* and *event sets* are defined, as well as the operations that are be performed upon them. Then, the necessary A-tree structures for efficiently performing these operations are described.

Finally, chapter 7 summarizes the findings of this thesis, and discusses the issues raised by this work which may become the object of future research.

1.2. The Music Data Manager (MDM)

A *music data manager* (MDM) provides a service to other programs, known as *clients*. The MDM delivers musical information, the client program uses it. For example, a music typesetting program would be a client, as would a musical score editor, a compositional tool, or a program which performs musicological analyses of compositions (figure 1.1). In current applications, these programs each are required to perform their own data management. They each have incompatible internal representations for the information they manipulate. Having a single MDM manage the musical information used by each of these clients provides certain benefits:

- The considerable burden, in terms of program complexity, of managing the data is no longer duplicated within every client.
- Any improvements or optimizations in the quality of data management provided by the MDM accrue to all its clients. Thus, optimizing one system causes improved performance in many systems.

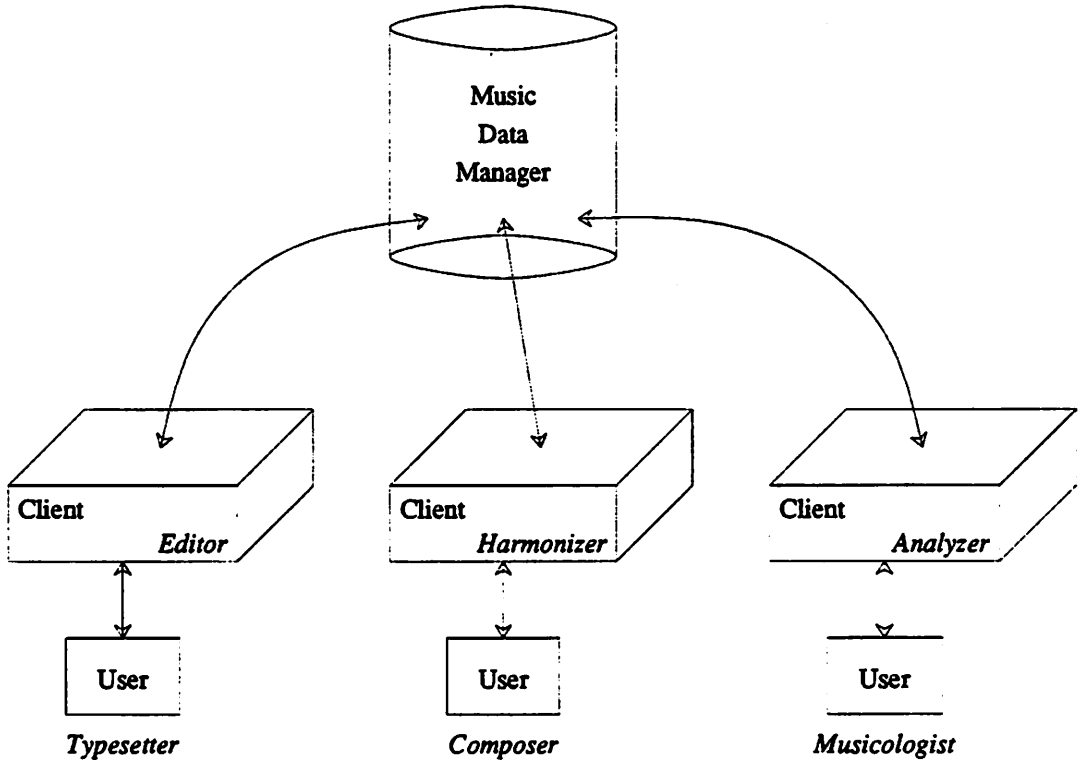


Figure 1.1. The Music Data Manager and Its Clients

- Because all clients maintain their information in the same way, via the MDM, they can more easily communicate with each other. For example, a music analysis program can easily process the output of a composition program, if both have been designed to use the same MDM.
- A good data model within the MDM should allow the development of clients that are faster to implement and easier to maintain, because the client need only manipulate a high-level musical information abstraction.

The MDM must handle standard database operations, such as concurrency control and recovery, as well as those particular to the musical domain. The primary extensions to traditional database systems considered by our research pertain to the modeling of music semantics, and the implementation of structures to support the physical realization of that model. This requires that some decisions be made *a priori* as to what type of clients will be served by the MDM. The following candidates are considered:

Editors and typesetters: These systems, such as SCORE [Smi72], MOCKINGBIRD [MaO83], and SMUT [Byr84], usually manipulate some form of musical *score*. They are highly interactive, and they retrieve, modify, and generate musical information. Clients of this type are typically concerned with a single musical work at a time.

Compositional Tools: Like editors, these systems are generative: they produce music (often in both sound and graphic representations). A number of computer languages and paradigms for music composition have been developed (see [LoA85] for a survey). A compositional tool might retrieve compositions written in these languages from the MDM, play them, modify them, and update the database.

Score Libraries: Large collections of musical scores, often containing the complete works of a given composer or era, serve as the starting point for most musicological research. Like most information retrieval systems, they must provide rapid data retrieval, but modification of the data is relatively rare. In practice, these computerized libraries are often highly selective. For example, they may contain only bibliographic information (as do most text based systems), or only incipits (opening melodies) rather than complete scores (as in an incipits database of Renaissance polyphony [Lin77]). A current index of computer assisted research in musicology [HeS86] lists twenty-five projects related to thematic indices, and eleven projects involved in collecting full musical scores. Each of these projects uses software specifically designed for its own application.

Music Analysis Systems: Music analysis involves applying particular operations to musical data. Systems that perform various sorts of harmonic analysis, or those that determine melodic structure are examples. Hewlett [HeS86] lists sixty ongoing research projects in music analysis. For musicologists, there are a variety of computer applications in this domain [Alp80, Gro84]. However, most of these research projects use custom designed programs. The musical information used as input to these analysis programs is typically not taken from a score library such as those described in the previous paragraph. Rather, the score data to be analyzed must be hand coded into an appropriate format for each analysis application.

1.3. Research Context

Because of the interdisciplinary nature of this dissertation, it draws ideas from several distinct bodies of research in both music and computer science. Rather than discussing all of them in detail here, each chapter will contain its own presentation of relevant research. However, in order to put this dissertation in perspective, a brief outline is presented of the related fields on which this work is based.

From computer science, relevant research has been conducted in the areas of database data modeling. Our starting point for a data model for music is the entity-relationship model [Che76], which in turn is an extension of the relational model [Cod70]. We have made use of other extensions to the relational model which are summarized in the RM/T proposal [Cod79]. These extensions have been considered for several application domains, such as statistical databases [Sho82], scientific databases [SOW84], pictorial databases [RoL85], and computer-assisted design (CAD) databases [SRG83].

Our proposals for managing attribute inheritance have benefited from research in the area of knowledge representation. Most knowledge representation languages, such as KRL [BoW77] and SRL [FWA84] address this issue. With respect to complex attribute inheritance, research in "idiosyncratic inheritance" [Fox79] has proven particularly applicable. Issues of inheritance, related to inference and deduction, have also been addressed in the database domain [ISW84, IoW85].

Many of the proposals in this paper stem from research on integrating abstract data types into the INGRES relational database system [Fog82, Ong82], particularly a data type representation for time [Ove82]. A proposal for incorporating user-defined aggregate functions over abstract data types has also proven directly applicable to our music representation problem [Han84].

In the music domain, this research is supported by previous work in the area of music representation. These include practical presentations of music notation [Don63, Rea69], as well as more theoretical analysis of notational systems [Wol77]. Score representations have been explored in the DARMS system [McL86a, McL86b], and in the composition/editing domain under the SSSP project [BRB78-BPR81]. Additionally, the field has seen research in artificial intelligence approaches to music representation [Roa79], and in expert systems [Ash83, Ash85]. Another recently proposed score representation [Dan86] incorporates versions and multiple views into its structure, relating to database

research in version control, as in [KaL82].

In both the database and music domains, the issue of managing temporal information has received considerable attention (see [BAD82] for a survey). This dissertation makes particular use of research in temporal modeling [And81, ShK86]. Time has also been considered specifically in music systems [DeK85, MaM70, Pru84b]. These systems are all concerned with the representation of temporal data, such as events and processes that transpire over time, multiple independent time lines, and virtual time.

CHAPTER 2

Musical Information

The information within the music manager incorporates several different facets of music, which we divide roughly into five categories,

- sound information,
- bibliographic information,
- "meta-musical" information.
- graphical information, and
- conceptual representations,

Each of these types of information will be discussed in this chapter, demonstrating the wide variety of types of information which must be integrated into the musical data manager.

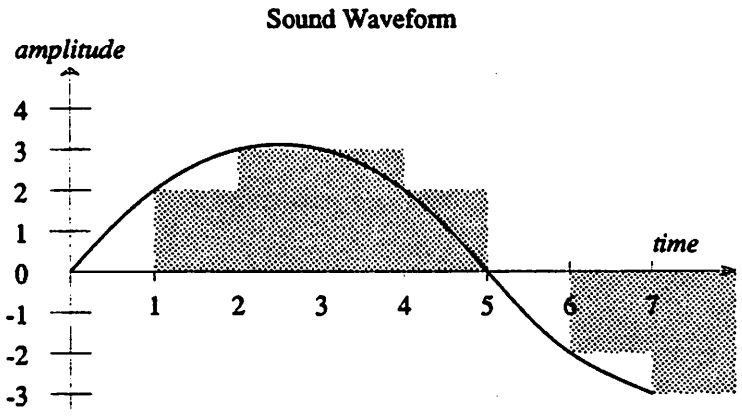
2.1. Sound Representations

Obviously, one fundamental type of object which a music information manager needs to represent is the sound of the music itself. The simplest representation of sound in a digital computer is merely an array of numbers, the result of *digitizing* the sound. Figure 2.1 shows an example of a simple method of digitization. On the left is a waveform represented by a curved line. Its amplitude is sampled at regular intervals, as indicated on the time axis. These samples form an approximation to the original waveform. This approximation is shown as the shaded region in the figure. The amplitude of the waveform at each sample point may be stored in a table, as shown at right.

The error associated with this digitized representation corresponds roughly to the difference between the line and the shaded region in figure 2.1. This error may be decreased by:

(1) increasing the rate at which samples are taken (the *sample frequency*), or

(2) increasing the precision of the sample values (for example, using 16 bit integers rather than 8 bit integers).



Digitized Waveform

time	*amp.*
0	0
1	2
2	3
3	3
4	2
5	0
6	-2
7	-3

Figure 2.1. Musical Information as Digitized Sound

Both of these measures increase the amount of data to be stored for a given piece of sound. Detailed analysis of this representation and its limitations may be found in signal processing texts, such as [OpS75].

Digital audio devices of professional quality typically use 16-bit integers for each sample, and record 48,000 samples per second of sound. This implies that ten minutes of musical sound can be recorded with acceptable accuracy by storing 57.6 megabytes of data.

Much research in audio signal processing analyzes methods for reducing this massive storage requirement while still preserving the aurally perceptible properties of the sound. From an information theoretic point of view, the digitized sound stream can be compacted in two ways: by eliminating redundant information from the sound stream, and by eliminating aurally imperceptible information from the sound stream.

Wilson [Wil85] surveys a number of these data reduction techniques of the first type, and Krasner [Kra79] discusses various encodings which are based on sound perception.

In contrast to random sound, or speech, music has a much greater burden of structure over and above that detected by these signal processing methods. This structure is what differentiates music from sound. For instance, rhythmic structure (e.g. a "beat") and timbral structure (e.g. that some sounds are generated by one instrument and some by another) may exist in musical sound. Such abstractions remain hidden at this level of representation.

The extraction of such structure, given only a sound representation, has proven to be quite difficult. Research by Chafe, *et al.* [CMR82], has found that a great deal of world knowledge (beyond that provided by the music at hand) is necessary to "understand" the structure of even the most simple musical pieces. Such knowledge might include an understanding of how the sound was produced (e.g. that individual notes are produced by particular instruments that constitute an orchestra), knowledge of the musical style or historical context of a piece (e.g. baroque or jazz), or knowledge of the performance practice of a piece (e.g. operas are performed with singers on the stage and an orchestra in the pit). Their research studies *automated transcription*, the generation of written musical scores, which exhibit much of the high level structure of a piece, from audio signals. Such work uses signal processing techniques to detect pitches and temporal distribution, followed by knowledge-based heuristic programming to detect musical structures, such as rhythmic constructs.

2.2. Bibliographical Information

An important use of music databases is as a reference for musicological research. Such a reference may provide several types of information. One common reference tool is the *thematic index*. Such an index is an organization of the works of a particular composer or period, including for each work sufficient musical (i.e. thematic) material to identify the composition. This is often a fragment of the melody or the key voices from the first several measures of the composition. Figure 2.2 shows a typical entry in a thematic index.

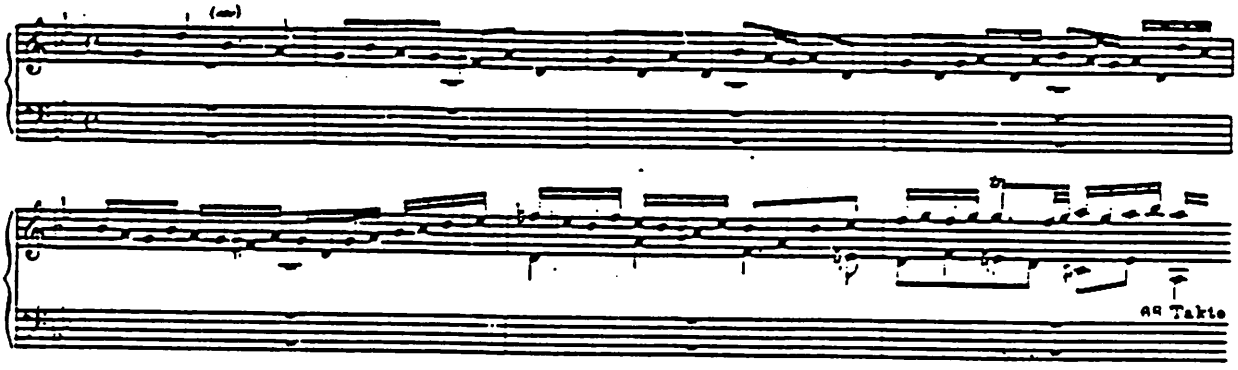
In addition to the thematic material that identifies the composition, several other pieces of information are provided in a highly compressed format. These are the orchestration or setting of the composition (*Besetzung*), when and where it was composed, how many measures (*Takte*) it contains, where copies (*Abschriften*) of the manuscript are located, editions (*Ausgaben*) in which it is printed, and articles written about it (*Literatur*). In the language of data management, these are each bibliographic *attributes* of the composition.

Once a bibliographic collection becomes established as definitive for a particular composer or body of music, the identifier created by the bibliographer may be widely understood to refer to a particular piece. Thus, the accepted name for the fugue in this example is as "BWV 578." "BWV" identifies the index (*Bach Werke Verzeichnis*), and "578" identifies the composition. In this particular

578 Fuge g-moll

Besetzung. Orgel.

BGA XXXVIII, 116. — EZ Weimar um 1709 (oder schon in Arnstadt?).



Abschriften. 2 Seiten im Andreas Bach-Buch (S. 65ᵛ—67ʳ), B Lpz. III. 8. 4. — In Konvolut quer 8° „aus Krebs' Nachlaß", BB in Mus. ms. Bach P 803 (S. 205—211). — Weiterhin in zahlreichen Einzelhandschriften u. Smlbdn. von der 2. Hälfte des 18. bis zur 1. Hälfte des 19. Jhs.

Ausgaben. In C. F. Beckers „Caecilia", Bd. II, S. 91. Veröffentl. nach e. Hs. vom Jahre 1754. — Peters Orgelwerke Bd. IV, S. 46. — Breitkopf & Härtel EB 3174, S. 72. — Hofmeister (Joh. Schreyer).

Literatur. Spitta I 399f. — Spitta VA 110. — Schweitzer 248. — Frotscher II 877f. — Neumann 51. — Keller 73f. — BJ 1912: 131; 1930: 4, 44, 126; 1937: 62.

Figure 2.2. A Thematic Index Entry [Sch50]

index, compositions are ordered chronologically.

Bibliographic information may be found in attributes at all levels of musical structure. At the highest level, compositions are placed in time (e.g. they have a "composition date" as an attribute), and are attributed to a composer. Individual sections of a composition may be borrowed from other composition, and thus they themselves may have a different composition date and composer. At the lowest level, the time at which individual notes are placed in a composition, and who placed them, constitute a form of bibliographic information. One might thus speak of a "micro-bibliography" as the internal history of a composition.

2.3. Meta-musical Information

Many of the "meanings" of musical information can be described either declaratively or procedurally. For example, consider the treble clef symbol. The meaning of this graphical icon might be described thus:

> All subsequent notes on the same staff as the treble clef have a mapping from staff degree to scale pitch which is "*E*very *G*ood *B*oy *D*oes *F*ine" (to use a favorite grade-school mnemonic).

This meaning can be interpreted declaratively, whereby all subsequent notes have the "treble clef" pitch interpretation, or procedurally, whereby the treble clef means that subsequent note heads are to be performed (or "mapped to pitches") in a particular way. In the first case, an icon determines a property of a passage. In the second case, the icon tells how to interpret the subsequent notes.

A more vivid example is provided by a musical *accidental* such as the sharp sign (#). A group of sharps placed at the beginning of a section of music composed in a particular style constitute a *key signature*. A key signature consisting of three sharps carries a declarative meaning, stating a fact about the *tonality* of the musical passage:

> The piece is in the key of A major (or f# minor).

It also carries the procedural meaning:

> Perform all notes notated as F, C, or G one semitone higher than written.

Much of the information contained in the music database may be derived procedurally from other declarative data in the database. Suppose that the database contains, as part of a score representation, a note object. An attribute of this note would be the staff on which the note lies. Another attribute would be the performance pitch of the note. However, the performance pitch of a note depends procedurally (as in the above two examples) on other elements on the same staff line, such as clefs and key signatures. In fact, there are other pieces of information, such as stylistic information about a composition, which govern the interpretation of performance pitch from graphical criteria. These rules constitute "meta-musical" information, and are part of the musical data to be maintained with the score.

This issue of maintaining procedural information will be explored in more detail in chapter 3, in the discussion on complex attribute inheritance.

2.4. Common Musical Notation

In the case that the "listener" of a piece of music is a person (as opposed to a recording device), raw audio information is in general not sufficient for the recipient to fully understand the performance. For example, the following operations, related to the transcription of sounds into scores, are difficult for human experts to execute.[1] Given an audio representation (e.g. a recording) of a piece of music:

- Determine the rhythmic structure of a composition that contains multiple independent voices.
- Determine what pitches are being played, in the face of complex harmonic structures.
- Determine what instruments (even assuming they are familiar to the listener) are performing which musical events.

A useful written notation for music conveys the above information clearly from composer to performer, along with additional information which is similarly obscured in the audio representation.

Music, like natural language, has many written forms which developed slowly over time along different paths within different cultures. Although there is no universal written musical form, there is a reasonably well defined language of music notation which has been codified for Western tonal music used from about the 17th century to the present. We will refer to this as *common musical notation* (CMN).

As a "language" of musical notation, CMN has its grammatical rules. These may be found in standard textbooks [Don63, Rea69]. More exacting notators, such as engravers who print music, require more detailed graphical information such as is presented in [Ros70].

Consider the score page as a purely graphical construct, that is, as uninterpreted black shapes (graphical objects) on a white page. We can collect the graphical objects in a CMN score into a "font," analogous to a font of alphabetic characters. Representations for such fonts are included in specifications such as PostScript [Ado85] and Metafont [Knu86]. Attempts have been made to create music fonts in this same manner, as in the Symphony™ font [Hug86]. In the course of our work, we have developed our own music font, which is outlined in appendix A.

---

[1] The difficulty of these operations is one of accurately interpreting perceptual data. The problems associated with this interpretation are common knowledge among those who have experience in music transcription. For this reason, we may speak of music transcribers as "experts."

Most graphical objects in the musical score fall into two categories, iconic and linear. Icons are graphical objects which have a particular shape, and which can logically be scaled to a larger or smaller size. Icons also have a fixed orientation. Figure 2.3 demonstrates these transformations. The characters in "alphabetic" fonts, in contrast, are not typically linear. They are only iconic in nature (i.e. alphabetic characters in a particular font are not usually subjected to stretching or rotation).

Linear objects (or linears, for short) have, in addition to their particular shape, an axis along which they are aligned. In general, given two points in the plane, a linear object can be stretched between them. Examples of linears are bar lines (whose axes are vertical) and staves (whose axes are horizontal). Certain linears may be rotated arbitrarily. Rotating linears include certain slurs and beams. These examples are depicted in figure 2.4.

With very few exceptions, all CMN elements can be represented with either icons – defined by shape, size, and position; or linears – defined by shape, size and the positions of the two endpoints. An example of an exception would be the less common complex slur, which can follow a somewhat arbitrary path (see figure 2.5).

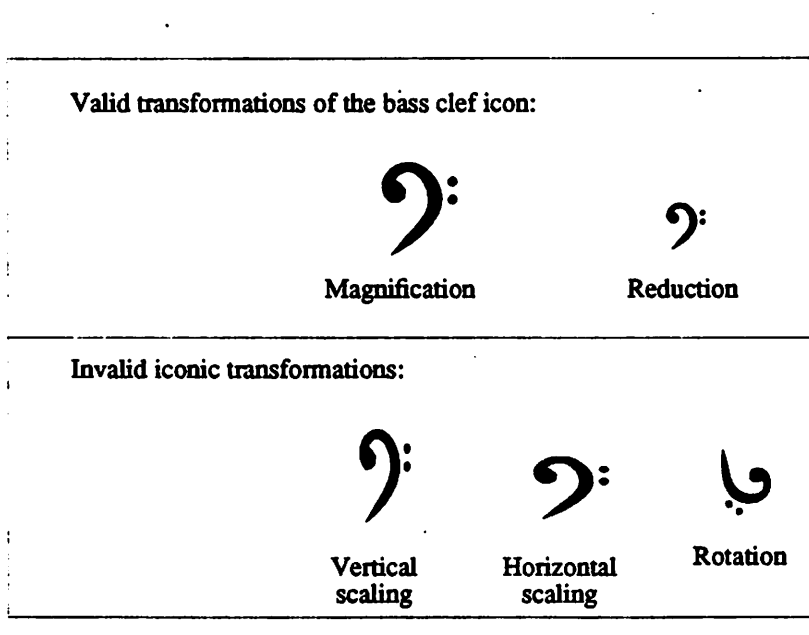


Figure 2.3. Iconic Graphical Object

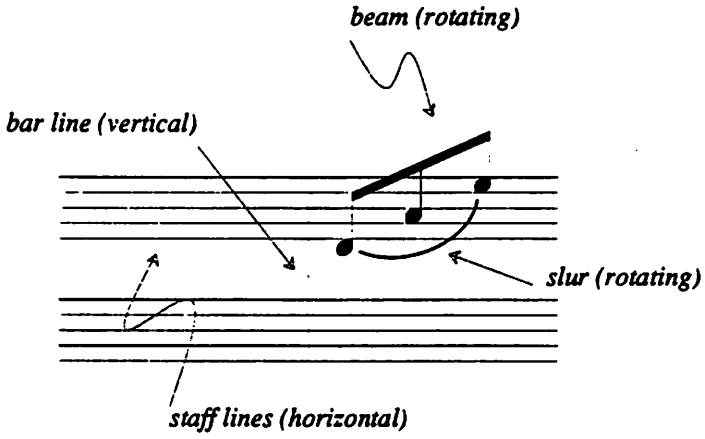


Figure 2.4. Linear Graphical Objects: Horizontal, Vertical and Rotating

Figure 2.5. A Non-linear Slur

Figure 2.6 summarizes the graphical transformations that may be applied to each of the different types of graphical objects. Organizing the graphical objects of a CMN score in this way greatly simplifies the process of representing such a score. Rather than specifying each graphical object completely, we need only refer to its type by name, and supply its relevant parameters (e.g. location and scale).

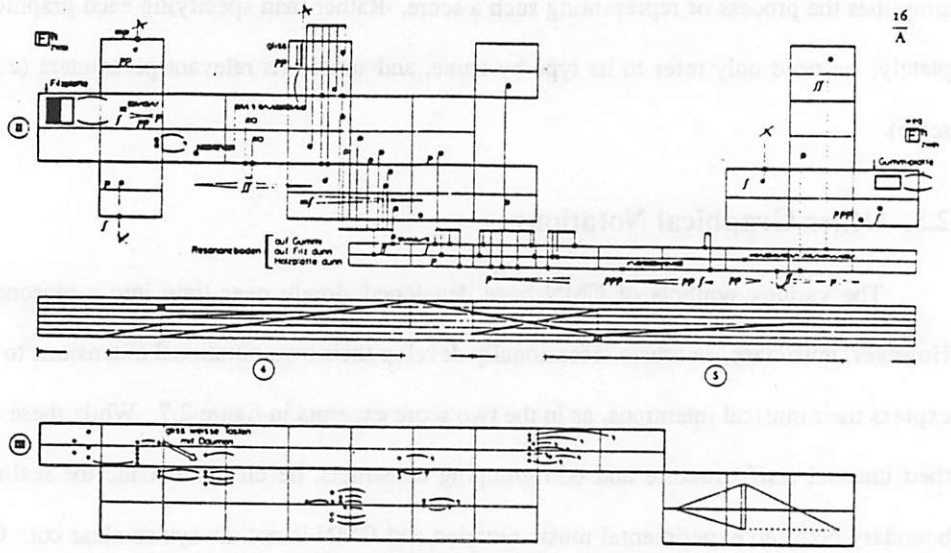
2.5. Other Graphical Notations

The various symbols of CMN have developed slowly over time into a reasonably stable set. However, musicians, as artists, occasionally develop their own notational extensions to CMN to better express their musical intentions, as in the two score excerpts in figure 2.7. While these examples, with their unusual staff structure and odd grouping constructs, lie clearly outside the realm of CMN, the boundary between experimental music notation and CMN is not always so clear cut. Over time, new notations become common practice and enter the realm of CMN. Many of these marginal notational practices are covered in [Rea69].

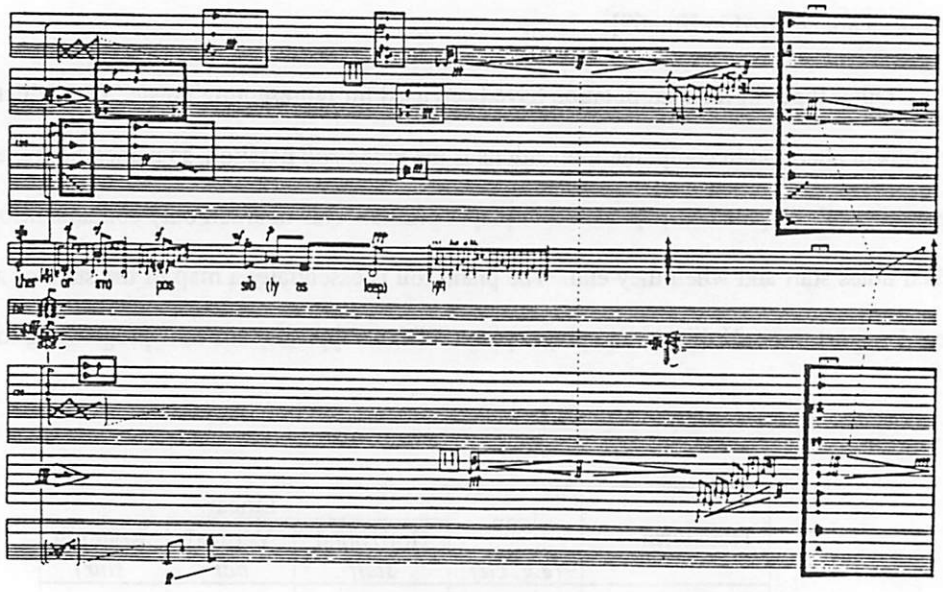
Other types of musical notation have been used for representing music. One which has received prominent attention in computer applications is the *piano roll* notation, so named because it looks similar to the rolls of punched paper used in player pianos. This notation only contains information about when notes start and when they end. The piano roll is essentially a map of the state of a musical keyboard against time. Unlike actual player piano rolls, we typically see time progressing to the left along

Transform	Icon *(e.g. clef*	Linear		
		Horizontal *staff*	Vertical *bar*	Rotating *slur)*
Translation	•	•	•	•
Magnification	•	•	•	•
Reduction	•	•	•	•
Horizontal scaling		•		•
Vertical scaling			•	•
Rotation				•

Figure 2.6. Possible Transformations for Graphical Objects



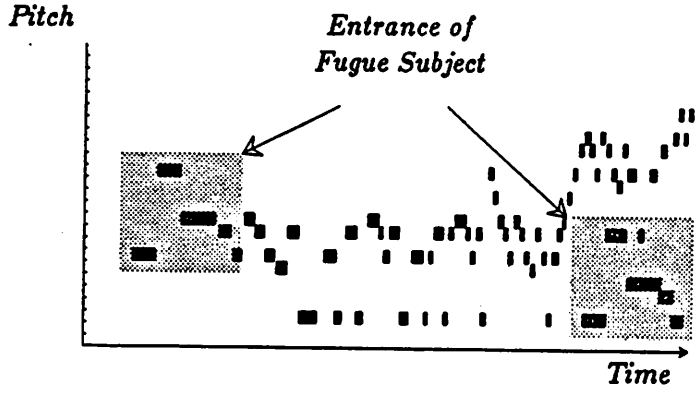
Page 16A of *Transicion* by Kagel
(in [Kar72, p. 105])



Page 24 of *Circles* by Berio
(in [Kar72, p. 89])

Figure 2.7. Non-standard Musical Notation [Kar72]

the x-axis, and pitch (usually quantized by semitones) increasing upward along the y-axis. Figure 2.8 shows an example. At the top of the figure is a piano roll representation of the first six measures of a Bach fugue [Bac47]. The CMN score for those measures is shown at the bottom of the figure. Each



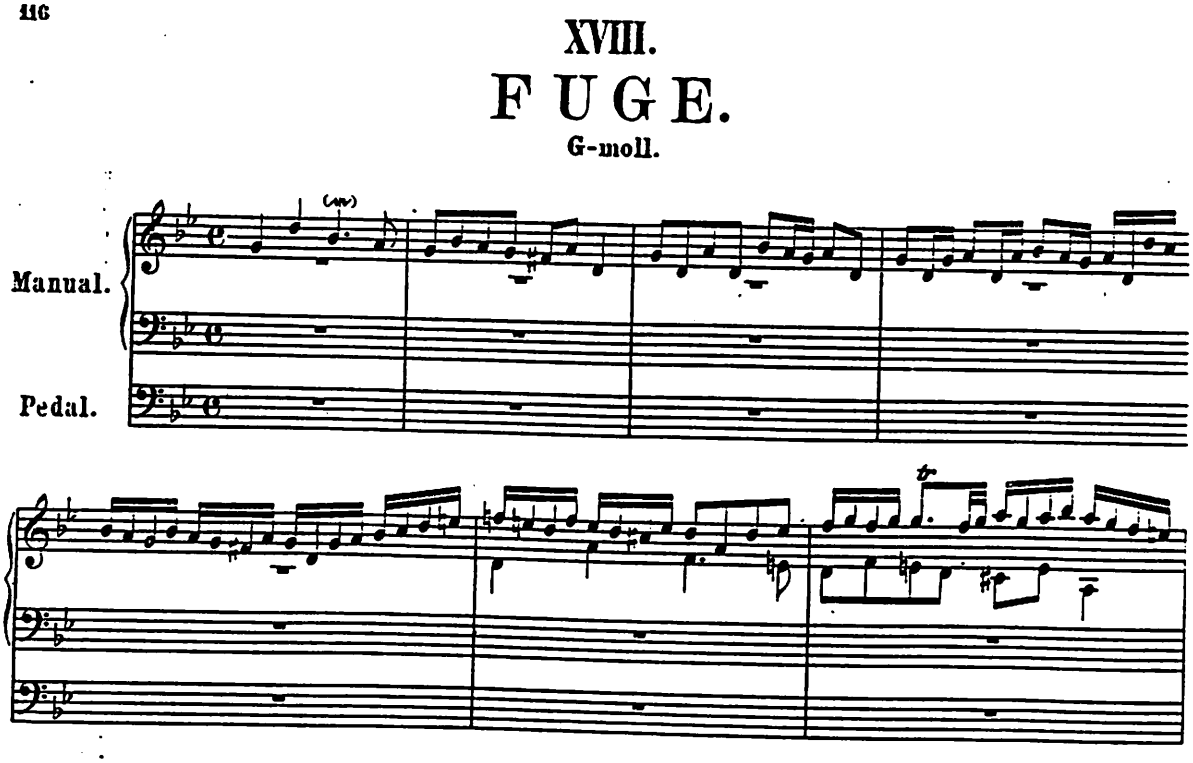


Figure 2.8. A Piano Roll

note is represented by a black rectangle. The entrances of the fugue, which are normally hidden in a piano roll notation, have been shaded in grey. They are clearly distinguished in the CMN score below by a change in note stem direction.

Some existing systems have been developed to edit and display piano rolls, such as SCORED [BSR79] and INTERSCORE [Pru84a]. The popularity of piano roll notation is explained by the ease of translation between note event streams (as generated by a variety of electronic music keyboard products) and piano rolls.

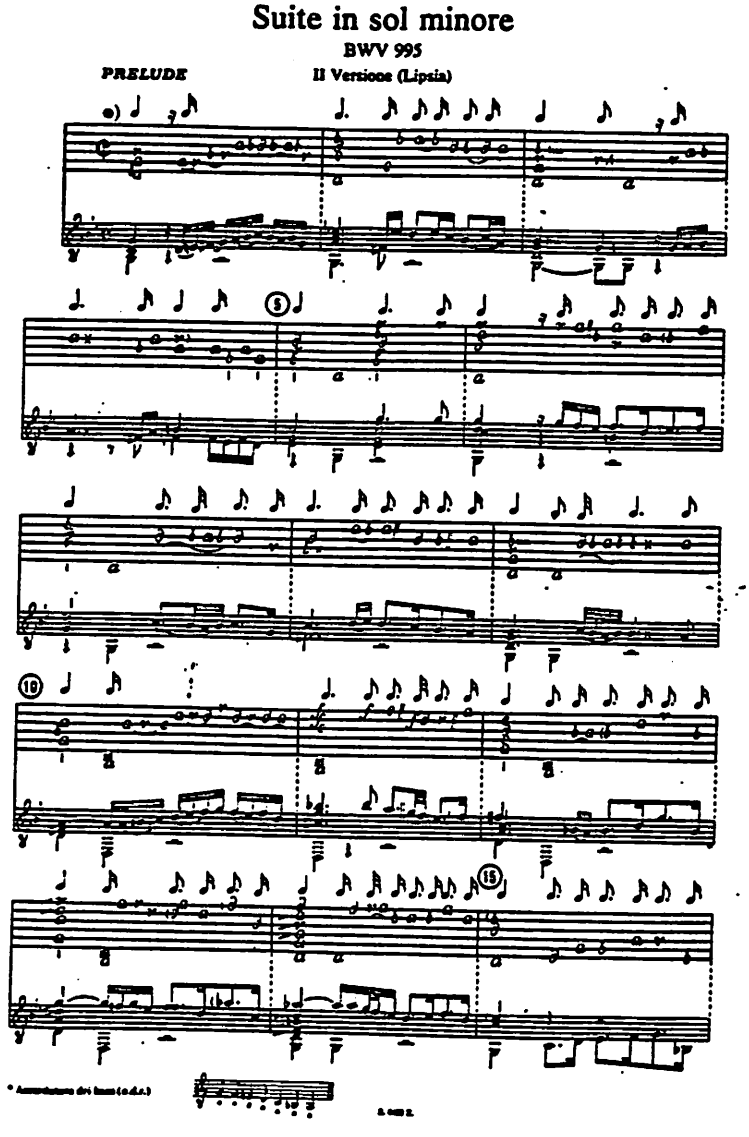
Other graphical notations have been developed. These are typically oriented towards specific instruments, as in the case of tablature notation for fretted instruments such as the lute, shown in figure 2.9. In this notation, each of the six staff lines represents one string on the lute. Another example is Equitone notation, developed more recently as a replacement for CMN (unsuccessfully, at present). Figure 2.10 shows an example of this type of notation.

2.6. Encodings for Representations

Before discussing how these various representations are to be encoded for the data manager, let us consider the various levels at which such an encoding may be done.

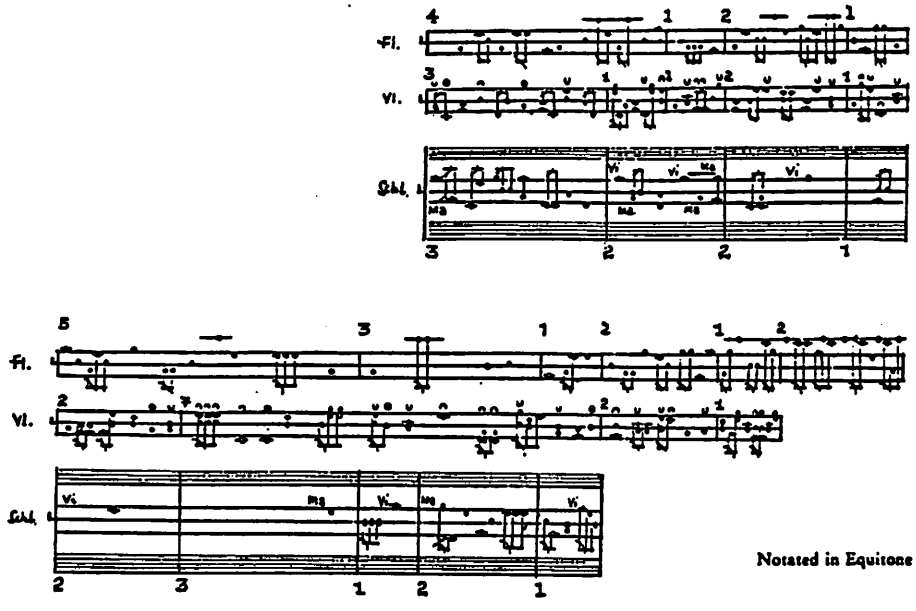
A number of layers of semantic abstraction can be formed for musical information. At the lowest level are uninterpreted bit streams, both digitized sound or raster graphics. Figure 2.11 summarizes one possibility for the various layers which can be built on top of these. At each layer, an example of an existing representation language for that layer is given. These and similar representations will be discussed in the remainder of this section.

In the sound domain, music may be organized into event streams, as with industry standard MIDI (Musical Information Data Interchange) event lists [Jun83]. More abstractly, it may be represented by various programming language specifications, as in the CMusic system [Moo85]. In the graphical domain, the lowest level of encoding is simply digitized (raster) graphics. This can be abstracted into its constituent graphical shapes, icons and linears, and described using a graphical definition language such as PostScript [Ado85]. Finally, a CMN score constitutes an abstract representation of the graphical aspect of a piece of music.



Page 1 of *Suite in sol minore* (BWV 995) by Bach;
Lute Tablature with modern transcription by Paolo Cherici.
(in [Bac80, p. 12])

Figure 2.9. Instrument Specific Notation: Lute Tablature



Page E1 of *Enjambements* by Cerha
Figure 2.10. Equitone Notation [Kar72, p. 86]

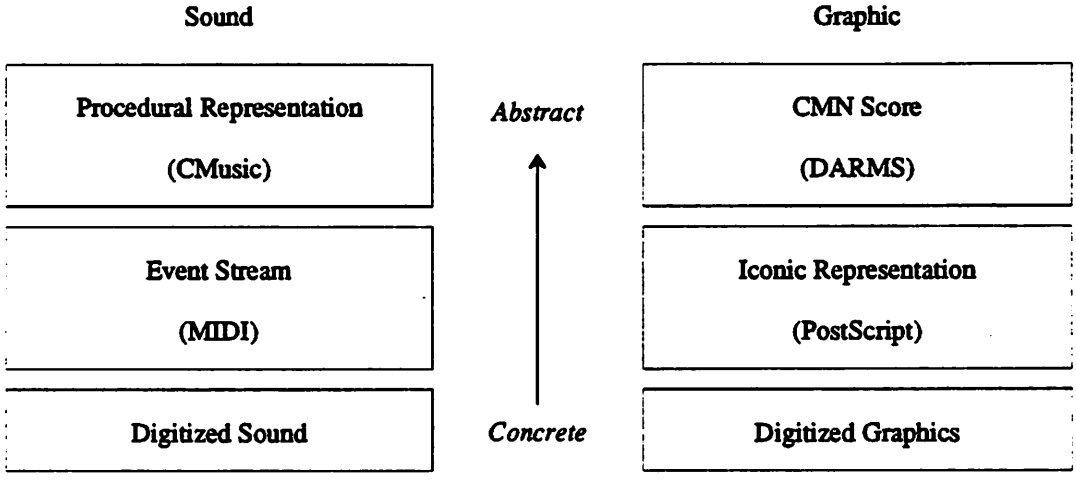


Figure 2.11. Layers of Conceptual Abstraction

2.6.1. Conceptual Representations of Sound

"Conceptual representations" are those representations of musical information which highlight the *structure*, or deep semantic content, of a musical composition. These tend to take the form of procedural languages for composition (similar to programming languages). Examples would be composition languages such as Music V [Mat69], CMusic [Moo85], or Flavors Band [Fry84]. Such languages allow the user to define voices, and the structure of the event streams in which these voices participate. For example, in CMusic, we can define voices to have (roughly) a particular harmonic contour, and a particular articulation (i.e. envelope). We then can define various sets of "notes" using control structures to indicate repeated sections, and so on. The resulting representation of a piece of music looks somewhat like a computer program, in that it has declarations, definitions, control structures, and statements. In this domain, the effect of executing a statement is (typically) to cause voices to sound.

Figure 2.12 shows a small sample of CMusic. Figure 2.12(a) shows a fragment of CMusic to define an instrument (i.e. sound generator), shown schematically in figure 2.12(b). This instrument is given a name in line 1, "flute". One oscillator (osc1 described in line 2) generates an envelope for the tone. The other oscillator (osc2 described in line 3) generates the waveform of the tone. Line 6 defines the waveform to be a sine wave, and line 7 establishes a typical envelope shape for osc1.

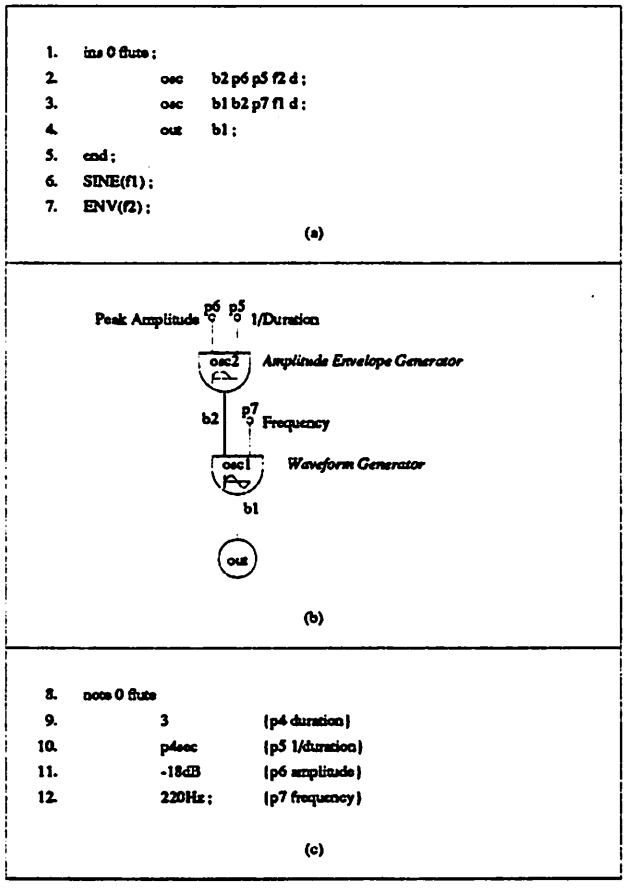


Figure 2.12. A Fragment of CMusic [Moo85]

After defining the waveform, a single note is played (figure 2.12(c)). The note statement in lines 8 to 12 defines the duration, amplitude and frequency of the note played on the "flute" instrument.

A MIDI (Musical Information Data Interchange) stream is an industry standard representation of music at the note or event level [Jun83]. The following description is slightly simplified. MIDI is based on a keyboard model, where each key is assigned a particular pitch, and keys can either be activated (down) or deactivated (up). The actual effect of keys going up and down is left to the device receiving the MIDI stream. A sample MIDI stream (for an "A major" scale) is described by figure 2.13. This particular example uses a protocol which puts time delays between events in the byte stream [Rol85]. The columns of the table are:

Delay (hex.)	MIDI command (hex.)			Time (secs.)	Command description
59	92	39	2E	3.625	A3 key on (channel 2)
54		3B	1F	4.325	B3 key on
08		39	00	4.392	A3 key off
3F		3D	2A	4.917	C#4 key on
0C		3B	00	5.017	B3 key off
3B		3E	22	5.508	D4 key on
05		3D	00	5.550	C#4 key off
50		40	10	6.217	E4 key on
0C		3E	00	6.317	D4 key off
3A		42	11	6.800	F#4 key on
0A		40	00	6.883	E4 key off
3C		44	27	7.383	G#4 key on
09		42	00	7.458	F#4 key off
48		45	2C	8.058	A4 key on
08		44	00	8.125	G#4 key off
48		45	00	8.725	A4 key off

Figure 2.13. Sample MIDI Stream

(1) The delay in time units (typically 1/120 second) from the previous event to this event. These are given in hexadecimal notation.

(2) A two or three byte MIDI command, typically representing channel, key, and volume. These are given in hexadecimal notation. A channel corresponds roughly to an instrument or voice. The key number is derived from a mapping of the piano keyboard onto consecutive integers. The volume integer ranges from one for most soft to 127 (hexadecimal "7F") for most loud, with zero reserved to indicate the termination of a note event. Only this column and the previous one are actually part of a MIDI byte stream, the other columns are explanatory.

(3) The time at which this event occurs, relative to the beginning of the event stream. This is therefore a cumulative event delay. It is given, in this table, in seconds.

(4) A description of the effect of the MIDI command.

This representation of musical events is not particularly general. For example, it has no way to represent events which are modified over their duration (such would be the case for a note which slowly grows louder over its duration). In fact, MIDI does not have the ability to represent non-point events at all. Notes must remain fixed over their duration; for example, once a note is turned on, its

volume or timbre cannot be changed under MIDI control, it can only be turned off.[2]

2.6.2. Conceptual Representations of Graphical Scores

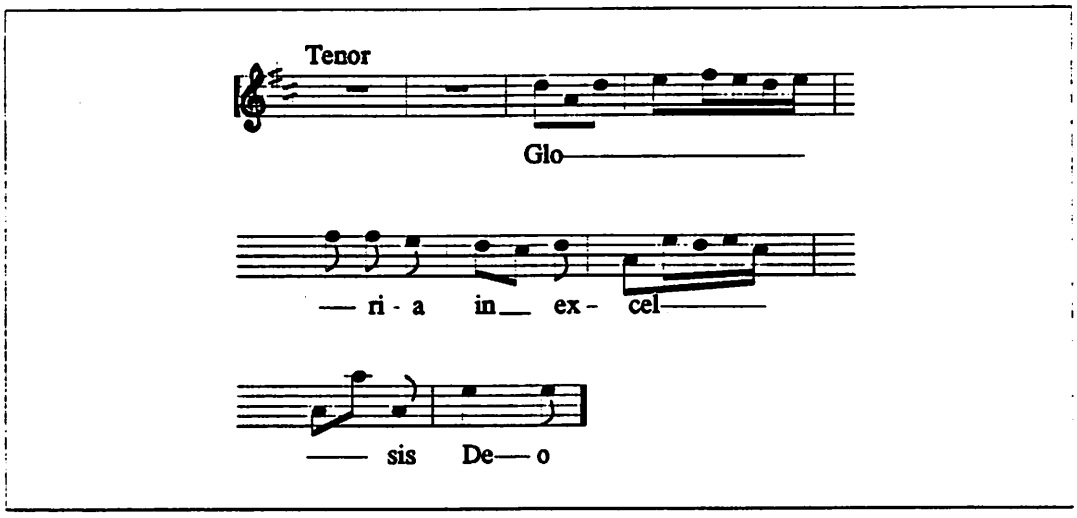
Now we turn our consideration to representations of graphical scores. Several methods have been developed to represent graphical scores in a form amenable to information storage and retrieval. Such systems include DARMS (Digital Alternate Representation of Musical Scores) [Eri77, ErW83], a general purpose encoding language whose goal is to objectively represent any score material notated using CMN. MUSTRAN [Wen77] is similar to DARMS, although its focus is on ethnomusicological material. Smith's system, SCORE [Smi72, Smi73] (now known as MS), is oriented toward producing very high quality graphical output. This system has interactive score editing tools that give the user very fine control over the music typesetting process.

As an example of these graphical languages, figure 2.14 shows a small piece of music, along with its DARMS encoding. This system was intended to encode musical scores onto punch cards (the project was started by Stefan Bauer-Mengelberg in the 1960's). It generally utilizes one letter codes for each attribute of an object found on the score. Numbers are used typically to indicate vertical position: 21 (or 1 for short) is the bottom line, 22 is the bottom space, and so forth. The other abbreviations are summarized in figure 2.14(c).

DARMS has a very flexible input protocol, allowing information to be entered from the page in a variety of orders (a measure at a time, whole lines at a time, etc.). Also, redundant information can often be suppressed, so that repeated note durations or pitches can be rapidly entered. Programs have been written to convert this "user DARMS" into "canonical DARMS" (the programs have been whimsically named "canonizers"). A canonical DARMS encoding presents the score information in a consistent order, and explicitly includes all repeated information [ErW83, McL86a]. Systems to generate a graphical CMN score from a DARMS encoding have also been designed [Gom77].

---

[2] In practice, this is not quite true. The timbral aspects of notes on a particular channel may be modified over time by MIDI commands that affect, for example pitch bend. However, these cannot be applied on a note by note basis. All notes active on a given channel are affected simultaneously.



(a) A Fragment of Music

```
I4 !G !K2# 00@¢TENOR$ R2W /
     (7,@¢GLO-$ 4 7) / (8 (9 8 7 8)) /
9E 9,@RI-$ 8,@A$ / (7,@IN_$ 6) 7,@EX-$ /
     (4D,@CEL-$ (8 7 8 6)) /
(4D 31) 4,@SIS$ / 8Q,@¢DE-$ E,@O$ //
```

(b) Its DARMS Encoding

Abbreviation	Meaning
I4	Instrument (or voice) definition #4
!G	G (treble) clef
!K	Key signature (!K2#: two sharps)
00	Annotation above the staff
R	Rest (two whole rests)
@*text*$	Literal string
¢	Capitalize next letter
(*notes*)	Beam grouping
W	Whole duration
Q	Quarter duration
E	Eighth duration
D	Stems down
/	Bar line

(c) Abbreviation Key for the DARMS Encoding

Figure 2.14. DARMS Encoding (from [Eri77])

2.6.3. Other Score Representations

Representations for music have been developed which are embedded into programming languages. An example of this is the LISP-based *Flavors Band* system developed by Fry [Fry84]. Flavors Band is intended for the procedural representation of jazz and popular musical styles. The system is primarily concerned with the pitch-time structure of a composition. In this respect it is similar to the MIDI specification [Jun83] in that it does not easily represent timbral or dynamic modifications to a single pitch over time.

PLA [Sch83] and Formes [RoC84] take an object-oriented approach toward representing the structural specification of musical scores. PLA is based on the text-based music representation, SCORE [Smi72]. Formes is written in LISP. Both provide a notion of *messages*, part of the object-oriented programming paradigm. A particular message is interpreted independently by each object type. Individual instruments of a composition may then respond to musical directives in their own way. For example, if a sound-generating instrument is sent the message *dolce* (sweetly), it might appropriately respond by decreasing its volume, lessening its vibrato, changing its timbral structure, and so on. A different instrument might interpret the *dolce* directive in an entirely different fashion.

PLA produces Music V note lists as output. Like Formes, PLA does not operate in real time. An entire specification is converted into digitized sound via several processes. This digitized sound cannot be played until all the processing has been completed.

A different approach is taken by the FORMULA system, built onto the FORTH programming language [AnK86a, AnK86b]. This real-time system supports algorithmic composition by allowing the user to manipulate multiple processes which independently schedule events (or attributes of existing events) over time. Events may actually produce sounds as they are scheduled interactively by the user.

2.7. Summary

In this chapter, musical information has been shown to consist of different types of data, including sound, graphics, text, and conceptual abstractions. Each of these data types has its own peculiarities of representation and manipulation.

Musical applications such as tools for score editing, composition, or analysis require the ability to manage these different types of information. A data manager also requires, in addition to information describing a particular piece of music, rules (what we have termed "meta-musical information") describing how the piece should be transformed, for instance, from its graphical form into sound.

Focusing specifically on CMN, we have seen how the graphical entities that constitute CMN scores may be divided into two graphical categories, iconic objects and linear objects. These together constitute the "font" with which CMN scores are notated.

Several encodings for both musical sounds and music notation have been developed. We have categorized these by their level of abstraction, with uninterpreted graphics data and digitized sound at the lowest level, and music descriptions such as CMN and programming languages at the highest level. In the next chapter, we consider structural characteristics of the more abstract representations, particularly of CMN.

CHAPTER 3

Hierarchical Ordering and Inherited Attributes

Because the music domain consists of well understood structural components (for example, CMN scores consist of staves, measures, notes, rests, etc.), the entity-relationship data model [Che76] provides us with a natural basis for describing musical information. This chapter begins with a review of the important features of the entity-relationship model. Essentially, each structure is represented in the database by an *entity*. In order to represent the relationships among these structures, we introduce the concept of *hierarchical ordering* as a tool for data modeling. We use three complementary representations for describing hierarchical ordering:

- *Instance graphs* as a pictorial representation of hierarchically ordered data,
- A *data definition language* (DDL) for hierarchical ordering,
- *Hierarchical ordering graphs* (HO graphs) to represent hierarchical ordering at the database schema level.

Section 3.1 begins with some background related to the use of ordering and hierarchy in database design. After introducing our representation of the entity-relationship data model, we present our extensions for hierarchical ordering.

In section 3.2, we consider an approach to representing inheritance in the music database. After a discussion of related proposals in other domains, we consider attribute inheritance in the music domain, and how it relates to these previous proposals.

A method has been developed to implement inherited attributes using a variant of the "query modification" technique used for maintaining *views* in a relational system. This will be presented in section 3.3.

There remains a type of inheritance that requires the full power of procedural specification to determine the values of inherited attributes. An extended example of this *complex attribute inheritance* is given in section 3.4. In this example, we present the inheritance procedures to determine the

"note volume" attribute for notes in a music database.

3.1. Adding Hierarchical Ordering to the Entity-Relationship Model

3.1.1. The Entity-Relationship Model

As a basis for the discussion which follows, we briefly review the entity-relationship model [Che76]. The domain to be modeled is represented by a variety of *entity types*. In the musical score domain, examples of entity types include compositions, measures, chords, notes, staves, and so on.

The actual objects within the domain are represented by *entity instances*. Each instance is of a particular type. Thus the composition entitled "The Star Spangled Banner" is an entity instance of type "composition." Every entity instance of a given type has a set of *attributes* associated with it. In the above example, "title" is seen to be one attribute of the "composition" entity type.

Within every entity instance of a particular type, each attribute is assigned a distinct *value*. Every entity instance of a given type has the same set of attributes, though the value of each attribute varies from instance to instance. For example, every composition is defined to have a title, and the value of that title is typically different for each composition.

A database *schema*, from our perspective, is the set of definitions necessary to describe the entity types in a database, their associated attributes, and the types of their interrelationships. Throughout this chapter, we develop a *data definition language* (DDL) to express them.[1]

3.1.1.1. Entities

An entity is defined by the **define entity** statement, whose syntax is:

```
define_entity_statement:
    define entity entity_name
        [ ( attribute_spec { , attribute_spec } ) ]

attribute_spec:
    attribute_name = attribute_type
```

The allowable "attribute_types" are determined by the implementation of this model, which will be

[1] DDL statements will be presented using BNF syntax descriptions [Bac59]. Key words will be given in bold face type, clauses surrounded by square brackets, [ ], are optional, and clauses surrounded by curly brackets, { }, may be included zero or more times. Upper and lower case letters are always distinct in key words (e.g. name, Name, and NAME are all different words).

covered in chapter 5. Generic types such as **integer** and **string** will be used in this chapter. Thus, we might define an entity type, "DATE,"

define entity DATE (day = **integer**, month = **integer**, year = **integer**)

A date entity has three integer attributes: its day, month, and year. Given this definition, we may manipulate a date as an atomic object, or we may refer to the individual attribute values within a particular date.

3.1.1.2. Relationships

A general description of relationships among entities is found in [Che76]. For the purposes of our discussion, we are interested in modeling two particular types of relationships, "*m* to *n*" relationships and "1 to *n*" relationships.

To express "*m* to *n*" relationships, we use the **define relationship** statement. Its syntax is similar to that of the **define entity** statement. Two (or more) entities are related by using their entity names as the types of the attributes of the relationship. For example, suppose that we wish to model compositions that are composed by many people (not common, to be sure). This represents an "*m* to *n*" relationship between people and compositions, because one person may be the composer of many compositions, and one composition may be written by many composers. This would be expressed as follows:

define entity PERSON (name = **string**, ...)
define entity COMPOSITION (title = **string**, ...)

define relationship COMPOSER (composer = PERSON, composition = COMPOSITION)

For a given instance of a COMPOSER relationship, the "composer" attribute references an instance of a person, and the "composition" attribute references an instance of a composition.

A "1 to *n*" relationship may be specified implicitly in the definition of an entity. Consider the relationship between compositions and their dates of composition. This is a "1 to *n*" relationship because an single date is associated with each composition, and an arbitrary number of compositions are associated with a single date. This relationship is expressed by the statement:

define entity COMPOSITION (title = **string**, composition_date = DATE)

A composition, as defined in this example, has two attributes. The first is a title, of type character string, and the second is the date on which the composition was composed. For a given entity instance of type COMPOSITION, the value of this latter attribute is a reference to some entity instance of type DATE.

Chen introduces a pictorial notation for representing entities and relationships, an example of which is shown in figure 3.1. This graph shows the definitions of PERSON, COMPOSITION, and DATE, and the relationships COMPOSER and COMPOSITION-DATE mentioned previously. In this type of representation, entity types are shown in rectangular boxes, and relationships are shown in diamond-shaped boxes. Lines are drawn from relationships to the entities which they reference. The type of the relationship (*m* to *n* or 1 to *n*) is indicated on these lines.

3.1.2. Ordering

Neither the relational model, nor the entity-relationship model incorporates any concept of ordering among elements stored in the database. Actual relational database systems, on the other hand, usually implement some form of ordering among data records. This is typically provided by allowing

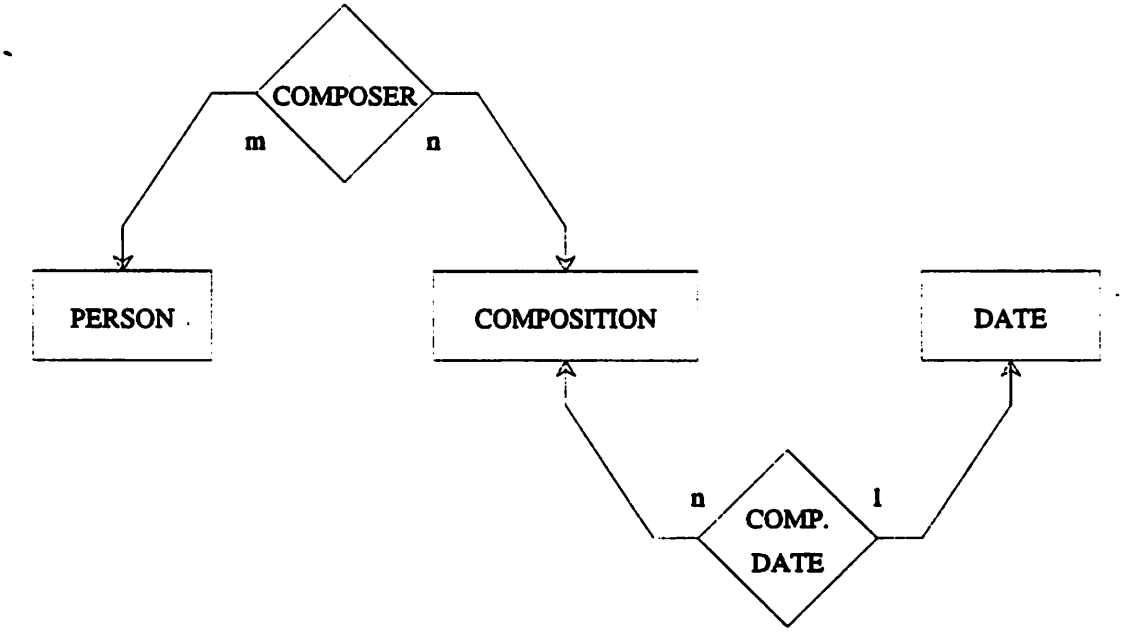


Figure 3.1. An Entity-Relationship Graph

the database designer to designate *key* attributes for a relation, allowing the system to sort the data records so that they are ordered by ascending (or descending) key value.

This use of ordering may be seen purely as a performance optimization in relational databases. An important relational operation is to select data records that have a particular key value (or range of key values). This may be efficiently performed on relations that are sorted, because the desired records are all stored together, rather than being randomly distributed throughout the relation.[2]

In contrast to this, we are interested in modeling a domain where an important attribute of the data is the participation of entities in various orderings. For example, a musical score consists of an ordered set of measures of music, and the fact that one measure follows another measure is a concept which must be modeled by the database definition.

We therefore extend our DDL with a statement to express orderings among entity sets. The syntax for the **define ordering** statement, in its simplest form, is:

define_ordering_statement:
    define ordering order_name (entity_name)

This represents the simple case where all the instances of an entity type participate in an ordering. For example, suppose compositions are ordered in the database according to their "importance." This is modeled by the statement:

define ordering IMPORTANCE (COMPOSITION)

Having defined an ordering such as IMPORTANCE, we will see later in this chapter how queries may be formulated to determine which compositions are more or less important than other compositions, and how the importance of a composition is fixed (at the time of insertion or modification) with respect to other compositions.

3.1.3. Generalization and Aggregation

The ability to model hierarchies has also proven important in the musical domain. Smith and Smith discuss two orthogonal types of hierarchy, generalization and aggregation, that apply to data modeling [LeG78, Smi72]. Their ideas were later implemented in database systems such as GEM

[2] Of course, this is only true if the desired selection is compatible with the choice of sort key. For instance, a relation

[TsZ84] and GAMBIT [BDR85].

Generalization hierarchies relate certain types of objects to *generic* objects. For example, an entity of type "boy" and one of type "girl" may both be related to the generic object "child". A child, in turn, is a specialization of the generic type "person". In the artificial intelligence domain, generalization hierarchies are sometimes known as *is-a* hierarchies [BoW77]. A girl *is a* kind of child, and a child *is a* kind of person.

Generalization hierarchies do not seems to be widely applicable as a tool for modeling musical information. For example, we do not find a musical object, x, to be a kind of musical object, y, which in turn is a kind of musical object, z.

In contrast to generalization hierarchies modeling the "kind of" relation, aggregation hierarchies model the "part of" relation. They provide a very powerful and expressive tool for representing such aspects of musical information as score structure. These hierarchies have many levels. For example, a note is a part of a chord, which in turn is a part of a voice, and so on[3]. In implementations such as GEM, an aggregation represents a fixed number of objects, each of different type, that combine to form a single aggregate object.

3.1.4. Instance Graphs

This notion of aggregation hierarchies must be altered in order to represent the ordered sets that occur in the music database. Unlike the aggregation hierarchies presented in [Smi72],

- the number of entities participating in an aggregation is not fixed by the schema.
- the entities participating in an aggregation form an ordered set. One may therefore speak of "the n-th entity" in an aggregation.
- All entities within an aggregation are typically (though not necessarily) of the same type.

The term *hierarchical ordering* will be used for this new form of aggregation. In its most general form, *hierarchical ordering* occurs when a group of database objects (of one or more types) forms an ordered set associated with a distinct parent object. For instance, a particular set of notes aggregate to

sorted on composition title cannot efficiently support a selection based on composer name.

[3] In this chapter, we will be drawing many examples for our data model from the musical domain. The various entities (e.g. notes, chords, and voices) will be defined more precisely in the next chapter. Readers already familiar with music notation may rely on their "common sense" understanding of the semantics of these musical objects.

form a given chord. An *instance graph*, such as the one in figure 3.2, shows this relationship pictorially. This graph, in its entirety, could represent, for example, a four note chord. It consists of a parent, y, and an ordered set of children, $\{ u, v, w, x \}$. The ordering among the children is indicated here by arrows from one child to the next one in the ordering. Such edges of the graph are called S-edges, as they indicate a relationship among siblings. Each child also has a relationship with its parent, indicated in the example by P-edges. Notice that each child has an ordinal position under its parent. For example, we may speak of the node w in this figure as the third child of the parent labeled y.[4]

Instance graphs represent actual pieces of data in the database, such as particular chords. In order to model chords in general, (i.e. what a chord *is*), the hierarchical ordering exhibited in the instance graphs must be defined in the database *schema*.

3.1.5. Defining Hierarchical Ordering in a Schema

In an aggregation hierarchy, the number and type of elements in the aggregation are fixed by the schema. For example, a piano is an aggregation of one keyboard, a fixed number of strings, a sounding mechanism, and a bench. A piano bench in turn is an aggregation of four legs and a cushion. For entities in the musical score, this characterization is insufficient. Specifically:

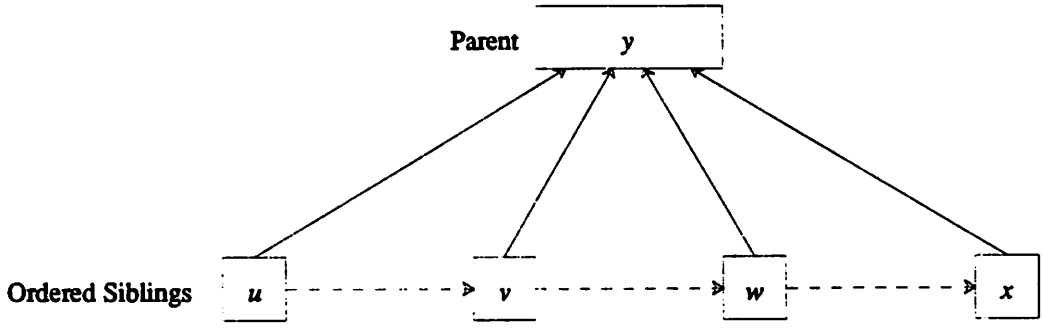


Figure 3.2. A Simple Instance Graph

[4] The pointers in an instance graph should not be misinterpreted as an indication of the physical implementation of these objects within the data manager. They merely serve to indicate graphically the ordering and hierarchy among objects.

- The number of objects in an aggregation is typically not fixed. For instance, under the aggregation of notes into chords, different chords typically have different numbers of notes.
- The objects in an aggregation are ordered. For instance, given two measures in a score, one must be prior to the other.

These characteristics distinguish hierarchical ordering from aggregation hierarchies. The **define ordering** statement is extended as follows to model hierarchical ordering:

```
define_ordering_statement :
    define ordering [ order_name ] ( child_entity { , child_entity } )
        [ under parent_entity ]

child_entity :
    entity_name

parent_entity :
    entity_name
```

One such statement defines a single instance of hierarchical ordering. "Order_name" is the name of the ordering. This is followed by one or more child relations whose instances will participate in the ordering. The **under** clause specifies the relation from which parent entities are taken, determining the type of the entity instance under which each ordering will be grouped. For example, a schema containing musical notes ordered within chords would be specified as:

define entity CHORD (*chord attributes...*)
define entity NOTE (*note attributes...*)

define ordering note_in_chord (NOTE) **under** CHORD

In this simple example, the ordering is named "note_in_chord." It consists of a single child type, NOTE, under the parent type, CHORD. This schema definition would allow reference to, for instance, "the third note in chord *x*."

The semantics of various forms of the **define ordering** statement, as when the order name is missing, or when there are multiple child types, will be the focus of the next section.

3.1.6. Types of Hierarchical Ordering

It will generally be more convenient to present ordering definitions in pictorial form. We therefore make use of the *hierarchical ordering graph* (HO graph), an example of which is shown in figure 3.3. This graph represents a single ordering. In general, each edge in the HO graph corresponds to

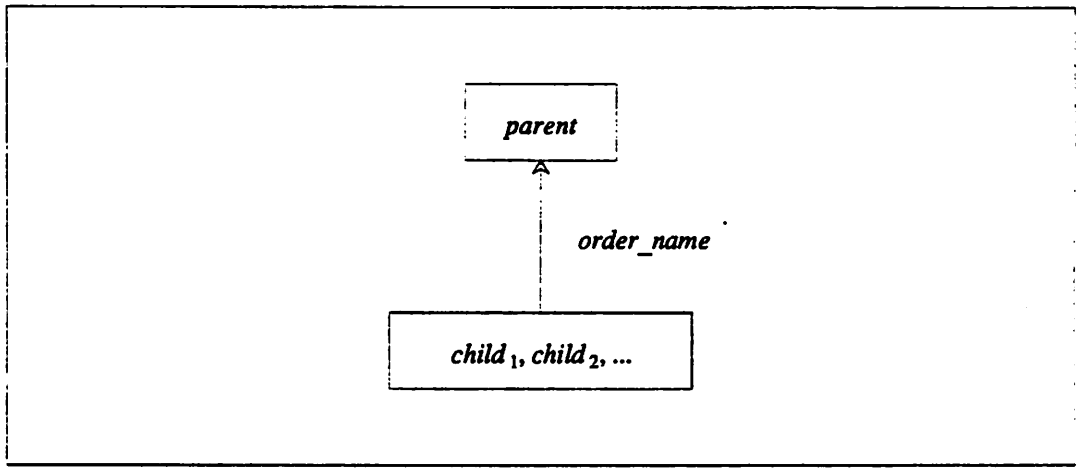


Figure 3.3. An HO graph for a Single Ordering

one **define ordering** statement.

We now consider several cases of hierarchical ordering.

Multiple Levels of Hierarchy. An object that is a parent in one ordering may be a child in another. Thus we may specify orderings in this way:

define ordering e (X) **under** Y
define ordering f (W) **under** X

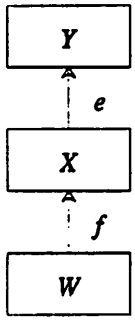
The HO-graph for this example is shown in figure 3.4(a). An instance graph is shown in 3.4(b). This type of ordering is quite common in music. For example, we might interpret this instance graph as representing notes within chords within a measure. Referring to the figure, the ordering *e* then represents the ordering of chords in each measure, and *f* represents the ordering of notes within chords.

Multiple Orderings Under a Parent. Figure 3.5 shows a slightly more complex case, where two different objects share the same parent, each under its own ordering. It is specified by two statements:

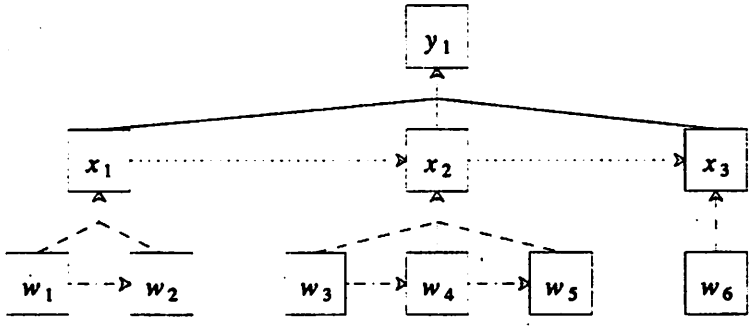
define ordering e (W) **under** Y
define ordering f (X) **under** Y

Figure 3.5(a) shows the HO graph, and figure 3.5(b) shows an example of an instance graph which would be possible under this schema. This type of ordering schema occurs, for example, where both

(a) HO graph:



(b) Instance graph:



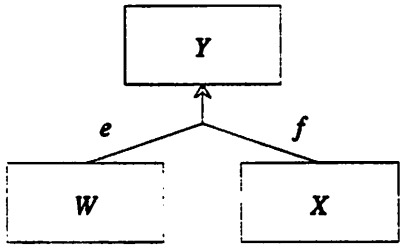
key:

P-edge for ordering e — P-edge for ordering f

S-edge for ordering e — S-edge for ordering f

Figure 3.4. A Hierarchy of Orderings

(a) HO graph:



(b) Instance graph:

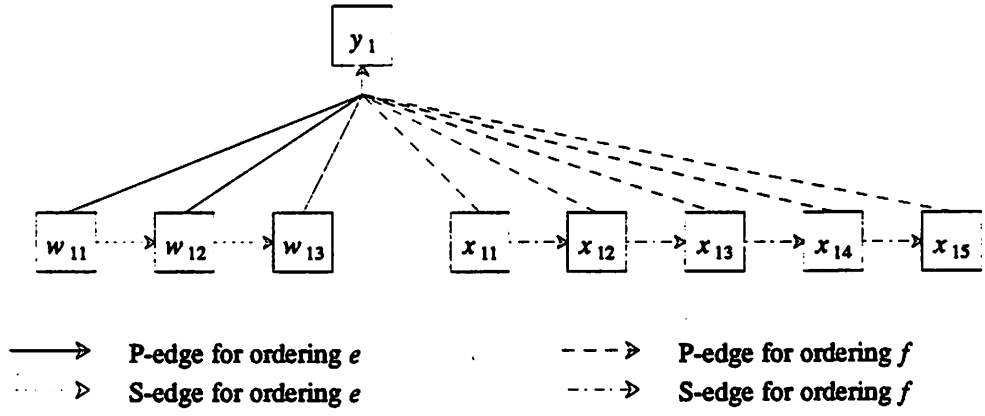


Figure 3.5. Two Orderings Under One Parent

parts and staves are ordered within an instrument (e.g. the portion of a score system dedicated to the violin instrument may contain three violin parts, notated on two staves). From this figure, *Y* represents the *instrument* entity type, *W* would be the *staff* type, and *X* would be the *part* type. The edges can then be named: *e* would be "the ordered set of parts per instrument" and *f* would be "the ordered set of staves per instrument."

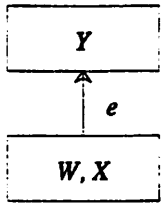
Inhomogeneous Orderings. The set of siblings in a particular ordering may not be of homogeneous type. Where two (or more) different types participate in a single ordering, we express their relationship by the single define statement:

define ordering e (W,X) under Y

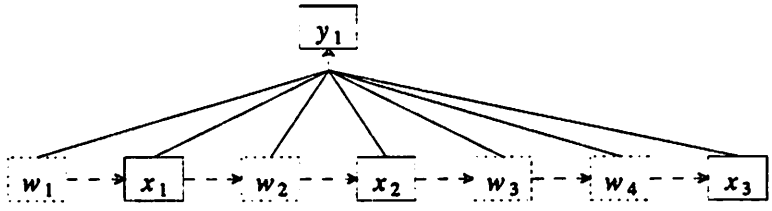
Figure 3.6 presents an HO graph that demonstrates this situation.

An example of this can be found in the music domain, where a musical *voice* consists of an ordered sequence of *chords* and *rests*, intermixed (this is a simplified view, for the purpose of this

(a) HO graph:



(b) Instance graph:



key:

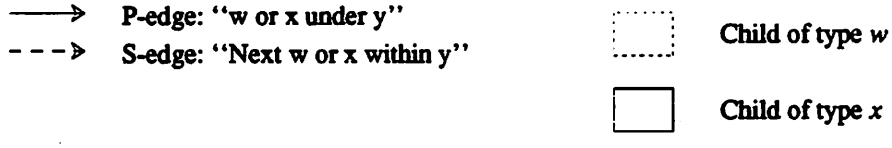


Figure 3.6. An Ordering with Inhomogeneous Children

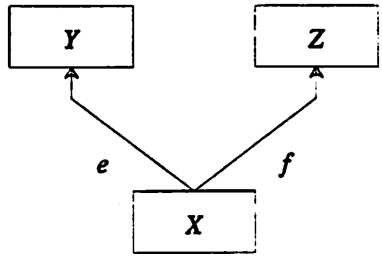
example). Every rest and chord, by our definition, has some voice as parent. The element at a particular position of the ordering, say, "the second object under voice V," must be either a chord or a rest. Of course, it can't be both, since there is only one "second object." This differs from the previous case, where a parent covered two child types under *different* orderings; then it made sense to speak of "the second part for the violin instrument" as well as "the second staff for the violin instrument."

Multiple Parents: Another possible configuration is for an entity to have multiple parents. Figure 3.7 shows the HO graph and instance graph for this definition:

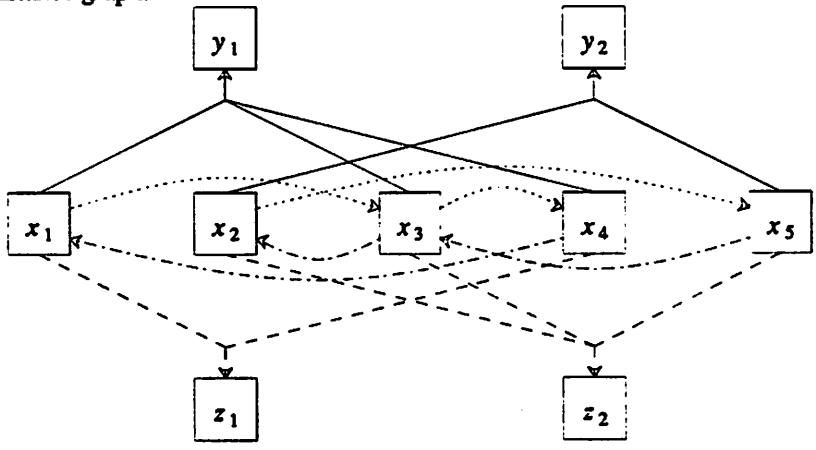
define ordering e (X) **under** Y
define ordering f (X) **under** Z

The HO graph in figure 3.7(a) shows an entity type X participating in one ordering e under parent Y and another ordering f under parent Z. A typical instance under this schema is shown in figure 3.7(b), while the table in figure 3.7(c) tabulates the ordinate position of each child node (of type X) under each of its parents (of types Y and Z).

(a) HO graph:



(b) Instance graph:



key:

——▸ P-edge: "x under y"
- - ▸ P-edge: "x under z"
·····▸ S-edge: "Next x within y"
·-·-▸ S-edge: "Next x within z"

	Ordinate Position of x_i			
Order name	*e*		*f*	
Parent	y_1	y_2	z_1	z_2
Child:				
x_1	1	-	2	-
x_2	-	1	-	3
x_3	2	-	-	2
x_4	3	-	1	-
x_5	-	2	-	1

(c)

Figure 3.7. An Entity Ordered Under Two Parents

In the musical domain, this multi-ordering structure is quite common. For example, a note has a chord as parent, under the ordering named "ordered set of notes per chord." A note also has a staff as parent, under the ordering "next note per staff". A chord may lie on multiple staves, so two notes that are members of the same "per chord" ordering are not necessarily members of the same "per staff" ordering.

3.1.7. Recursive Ordering

Suppose that the parent in an ordering is of the same type as one of the children. In that case, the ordering is recursive. An example from music would be found in the grouping of chords under beams. A *beam groups* consists of an ordered set of smaller beam groups intermixed with chords. This would be defined as follows:

define ordering (BEAM_GROUP, CHORD) **under** BEAM_GROUP

The HO graph for this ordering is shown in figure3.8(a). Figure 3.8(b) contains a fragment of musical notation with several layers of beam groups. The six chords in this fragment are labeled c_1 to c_6. The instance graph for the chords and beam groups is shown in figure 3.8(c). Every object in this instance graph is either a group (labeled g_i) or a chord (labeled c_i).

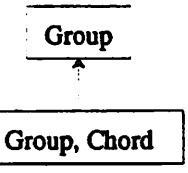
Certain restrictions on recursive ordering are necessary, to prevent the occurrence of instance graphs that are malformed. One difficulty arises if the P-edges for a given ordering form a cycle. Because this would mean that an instance is "part of" itself, such cycles in the instance graph are disallowed. Similarly, cycles among the S-edges of a given ordering are not permitted, because they result in the situation where an object is "before itself" in the ordering.

In the above discussion, several types of hierarchical ordering have been explored. We now consider the ways in which queries may be constructed that make use of the information provided by these orderings.

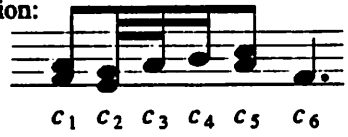
3.1.8. Manipulation of Ordered Entities

We use QUEL [Rel84] as a basis for our data manipulation language. Three new operators are added to QUEL to support hierarchical ordering: **before, after, and under.** Unlike other QUEL operators, the ordering functions operate on entities (represented by range variables in QUEL), rather than

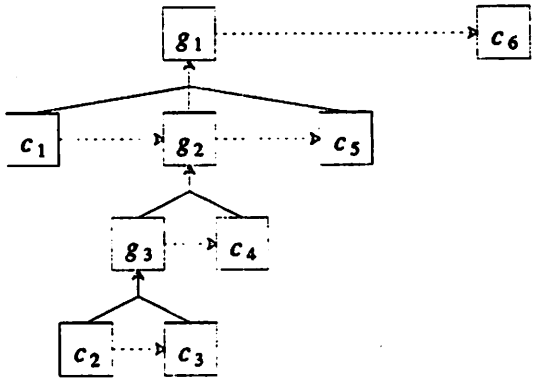
(a) HO Graph:



(b) Group/Chord Notation:



(c) Instance graph:



key:

⟶ P-edge: "Group or chord under group"
⋯▷ S-edge: "Next group or chord within parent group"

Figure 3.8. An Example of Recursive Hierarchical Ordering

on attribute values. In this way they are similar to the entity equivalence operator, is, introduced in the GEM extensions to QUEL [Zan83]. As an example of the is operator, recall the schema for composers and compositions:

```
define entity PERSON (name = string, ...)
define entity COMPOSITION (title = string, ...)

define relationship COMPOSER (composer = PERSON, composition = COMPOSITION)
```

A query to find all the composers of "The Star Spangled Banner" would be[5]:

```
retrieve (PERSON.name)
    where COMPOSITION.title = "The Star Spangled Banner"
    and COMPOSER.composition is COMPOSITION
    and COMPOSER.composer is PERSON
```

Unlike other operators, the **is** operator takes entities (i.e. range variables) rather than attribute values as operands.

The ordering operators each take two range variables and an optional ordering name as operands. The syntax for a qualification using the "before" operator is representative:

```
before_clause :
    range_variable before range_variable [ in order_name ]
```

The **after** and **under** operators have similar syntax. For the **before** and **after** operators, the types of both range variables are taken from the child types of the ordering indicated by "order_name." For the **under** clause, the type of the first range variable is taken from the children of the ordering, and the type of the second is the parent type in the ordering. The clause,

```
a before b in order_name
```

evaluates to "true" if *a* and *b* both have the same parent with respect to the hierarchical ordering indicated by *order_name*, and *a* is before *b* in that ordering. If *a* and *b* have different parents, then they are not comparable, and the **before** clause evaluates to "false."

Given these definitions of NOTE and CHORD,

```
define entity CHORD (name = integer, other chord attributes...)
define entity NOTE (name = integer, other note attributes...)

define ordering note_in_chord (NOTE) under CHORD

range of n1, n2 is NOTE
range of c1 is CHORD
```

here are examples of the use of the ordering operators:

Given a note *n*, retrieve the notes prior to *n* in its chord:

```
retrieve (n1.name)
    where n1 before n2 in note_in_chord
    and n2.name = n
```

Retrieve the notes that follow note *n*:

---

[5] As in GEM and later versions of INGRES, a range variable with the same name as its entity type is implicitly declared for each entity type.

```
retrieve (n1.name)
    where n1 after n2 in note_in_chord
    and n2.name = n
```

Retrieve the notes under chord *c*:

```
retrieve (n1.name)
    where n1 under c1 in note_in_chord
    and c1.name = c
```

Retrieve the parent chord of note *n*:

```
retrieve (c1.name)
    where n1 under c1 in note_in_chord
    and n1.name = n
```

3.2. Inherited Attributes

One of the principal motivations for organizing entities into a hierarchy is to allow attribute values associated with one entity to *depend* on those associated with another entity. For example, one attribute of a sync is its temporal location (i.e. the time at which the sync begins). All of the chords which belong to a single sync *inherit* this temporal location. We can thus refer to the temporal location of a particular chord, while guaranteeing that a set of chords defined to be simultaneous are indeed so (by virtue of their inclusion under the same sync). In this example, the temporal location of a chord is dependent on the temporal location of its parent sync.

An application such as a music typesetting program presents a more complex example. Suppose the application wants to query the music database:

- Given a note, *n*, what are the graphical coordinates (x, y) at which to draw it?

In a database that stored NOTE entities with position attributes x and y, this might be translated into the QUEL command:

```
retrieve (NOTE.x, NOTE.y)
  where NOTE.name = n
```

Storing graphical coordinates in the NOTE relation in this way suffers two serious drawbacks.

The first drawback is that important integrity constraints on positional information are missing from such a schema. Here are two examples of integrity constraints on the position of a note:

- Notes in the same chord must have the same x position.

That is to say that notes in the same chord must be aligned vertically.[6]

- The distance between the y position of a note and the y position of the staff on which the note lies (i.e. the y position of the bottom line of that staff) must be exactly a multiple of half the distance between staff lines.

In other words, a note head must sit on a staff line or a space, but nowhere else.

The second drawback is that storing the positional information of a note with the note itself is not robust in the presence of updates. For example, suppose that we insert a measure of music into a score. This must cause the position of all subsequent notes to change. Or suppose we perform the musically innocuous operation of moving a staff line slightly. Perhaps, for visual aesthetic reasons, we displace a staff line up or down on the page. The (x, y) position of every note on the staff must change because of that operation. There is no mechanism, within the database definition, to indicate that an update to staff position must cause an update to the position of several notes.

The root of the problem is that, although *position* is an attribute of a note, insofar as it is meaningful to ask "What is the position of note n?", the value of the *position* attribute is actually dependent on many other attribute values. For example, note position depends on note pitch, the vertical position of the staff on which the note lies, the horizontal position of the sync in which the note is a member (indirectly, via some chord), and the positions of other notes in the same chord (since the note may swing to one side or the other of the chord stem depending on the placement of other notes on the same chord stem).

When the value of an attribute is functionally dependent on the values of attributes in other entities, we say that this attribute value is *inherited.* The above example demonstrates *complex attribute inheritance.* The inheritance is considered complex because the attribute value is not simply the value of a similar attribute in some "parent" object in the database, but is rather an arbitrary function of a number of relevant attributes distributed among several database objects.

---

[6] Such an integrity constraint, as stated, is incomplete, of course. For example, it does not allow for the case where two notes in a chord differ by a single staff degree. In that case, the notes are on opposite sides of the stem, rather than vertically aligned.

Given the above description of inherited attributes, the following sections relate this form of inheritance to other types of inheritance that have been explored in previous research, and then present a proposal for incorporating inherited attributes into the music data manager.

3.2.1. Inheritance in Database Research

Existing research concerned with attribute inheritance, as it relates to the music database, falls into three domains:

- Artificial intelligence research focusing on knowledge representation,
- Database research involved in data modeling and data definition language design, and
- Music research focusing on the modeling of musical information.

Much early debate in artificial intelligence research focused on whether knowledge should have a procedural or a declarative representation, that is, whether knowledge should be represented by algorithms or by facts. The arguments for both sides of this issue are discussed by Winograd [Win75], who proposes a frame representation to capture the connections between various concepts. He focuses on inheritance of attributes in a generalization hierarchy.

The inheritance of attributes along a generalization hierarchy is very simple. If an object x is a specialization of another object y, then all the attributes of y are inherited directly to x. For example, if an attribute of the "person" concept is that a person is alive, then an inherited attribute of the "child" concept is that a child is alive (because a child is a person).

Later knowledge representations, such as KRL [BoW77], extend the semantics of frames to include distinctions between classes of objects (such as "person"), and instances of objects (such as "the person named Jane Smith"). They also permit a much more general specification of inheritance, allowing essentially general types of links between concepts. Still, in this system, inheritance is an implicit consequence of these links.

Fox [Fox79] proposed that inheritance should be separated functionally from the structure of the knowledge representation. In this proposal, inheritance itself is a concept to be modeled. Inheritance concepts take different forms, each with its own attributes (which determined the specific type of inheritance). The inheritance displayed by generalization hierarchies (*is–a* inheritance) is one exam-

ple of such an inheritance concept. He uses the term *idiosyncratic inheritance* to describe the arbitrary forms of inheritance that do not fit into predefined inheritance classes.

His language for determining the nature of such an inheritance specifically includes constructs for determining what attributes are to be *passed* to the inheriting concept, and what attributes are to be *added, excluded, contradicted, restricted, refined* or *generalized.*

This approach has the power of specifying arbitrary types of inheritance, while still maintaining the succinctness of established inheritance classes, such as that of generalization hierarchies. This form of inheritance specification was incorporated into the SRL system [FWA84]. However, this system still does not address the problem of complex inheritance, for instance, it does not provide a means to inherit, say, the largest value from a set of related objects.

A number of systems and languages have recently made use of the object-oriented paradigm (see [StB86] for a survey and introduction). In these systems, such as SMALLTALK [GoR83] and LOOPS [BoS81], entities known as *objects* contain both procedural and state information. The objects are manipulated by sending messages to them. Objects are typically divided into classes and instances. For example, "person" is a class, and "Jane Smith" is an instance of that class.

Every instance inherits the properties of the class of which it is a member. All the properties of persons in general are properties of "Jane Smith". In addition, classes are organized into a generalization hierarchy, and so instances further inherit the properties of all classes that are superordinate to the class of persons (i.e. the class of mammals, and the class of animals).

From our perspective, there are two advantages to the object-oriented paradigm. First, the functional separation between classes and instances accurately models the music database distinction between entity types and entity instances. Secondly, the notion of procedural attachment, that is, associating procedural information with any class object in the generalization hierarchy, provides the ability to handle arbitrarily complex computations to determine the value of an attribute (at the time an instance of the class receives a message to produce that value).

The disadvantage of this approach is that by restricting inheritance to the paths of the generalization hierarchy, it is insufficiently powerful to model those cases where classes are organized as aggregation hierarchies, as in hierarchically ordered data.

When an instance receives a message, it checks to see if the class of that instance knows how to respond to the message. If not, it passes the message on to its parent in the generalization hierarchy. In a sense, this "passing on" is the operative part of this inheritance. A limitation of object oriented systems is the restriction they place on which classes provide the information to respond to messages.

Recent research in database management has looked into issues related to knowledge representation and inheritance. A number of proposals, for example [DeF84], have endeavored to make use of the data management services of the database in order to efficiently handle a large body of static world knowledge. Others have taken the object-oriented approach and considered the problem of data management of objects, for example GEMSTONE system built onto SMALLTALK [CoM84] and the EXODUS system [CDR86].

A knowledge base typically contains a very large number of rules and procedures. For a given task, only a small number of those rules may apply. One of the primary problems faced by knowledge-based systems is the problem of efficiently determining which rules are relevant to a query. Much of the impetus for using database systems to store procedural information is the attractive possibility of using sophisticated data management techniques to perform efficient selection of rules [SSH86].

The role of artificial intelligence in music has been surveyed by Roads in [Roa85] which contains a large bibliography of relevant literature. Four specific application areas are noted in this survey: composition, performance, music theory, and digital sound processing.

The use of production (i.e. rule-based) systems is particularly interesting to us insofar as their process of inference closely models the complex attribute inheritance we wish to capture. Ioannidis has suggested extensions to QUEL to support production systems [ISW84]. Such systems have been used for both composition, and musical analysis in the music domain. The general application of automatic composition systems is surveyed by Hiller [Hil70]. Actual systems include a production system employing Schenkerian synthesis to generate four-part chorales by Ebcioglu [Ebc84] and a similar system using more general synthesis rules designed by Thomas [Tho85]. Rule sets for performing phrase structure analysis are given in the context of a general production system in [Ash83, Ash85].

3.2.2. Representing Inherited Attributes

The set of attributes associated with an entity may be divided into two types: Those whose values are *native* to the entity instance, and those whose values are *inherited* from some other entity. If it makes semantic sense to update the attribute for a given instance, while leaving the instance graph unchanged, then the attribute is native. Thus an attribute such as "stem direction" is native to a chord. All the connections to and from the chord may remain unaltered in the instance graph if the stem direction changes from "up" to "down." Temporal location is not a native attribute of a chord, since changing the temporal location of a chord necessitates moving it from one sync to another.

Native attributes are associated with entities at the time they are defined with the **define entity** statement. For example, the native attribute for stem direction is incorporated into the definition of the CHORD entity as follows:

define entity CHORD (... stem_direction = **string** ...)

We extend the syntax of our DDL as follows to provide a definition for inherited attributes:

define_inheritance_statement:
    define inheritance entity (target_list)
        where qualification

entity:
    range_variable

target_list:
    attribute_name = expression { , attribute_name = expression }

The **define inheritance** command adds additional attributes to an existing entity. The value of an inherited attribute is the value of the expression associated with the attribute by this definition, evaluated at the time the value is accessed. The syntax of this statement is similar to that of the **replace** command in QUEL, but rather than replacing an attribute value, the **define inheritance** statement adds new attribute values to the entity associated with the given range variable.

In the presence of hierarchical ordering, we can divide inheritance functions into three broad categories: downward inheritance in the hierarchy, inheritance from ordered aggregation, and upward inheritance in the hierarchy. We will now consider examples of inheritance taken from each category.

Downward Inheritance in the Hierarchy: Attribute values may propagate down the hierarchy. A child thus inherits attributes of its parents (and recursively, of its ancestors). In this way a note inherits temporal location (i.e. the *start_time* attribute) from a chord. This would be defined as follows:

```
define entity CHORD (start_time = i4, other native chord attributes...)
define entity NOTE (native note attributes...)

define ordering note_in_chord (NOTE) under CHORD

define inheritance NOTE (start_time = CHORD.start_time)
    where NOTE under CHORD in note_in_chord
```

Inheritance from Ordered Aggregation: A child under a given parent may inherit attribute values that depend functionally on the set of siblings of which the child is a member, and on the position of the child within that ordered set. A measure, for example, has the attribute *measure number*, which is a count of the number of measures preceding it in its ordering under a given movement. This could be specified by:

```
define entity MOVEMENT (native movement attributes...)
define entity MEASURE (native measure attributes...)

define ordering measure_in_movement (MEASURE) under MOVEMENT

range of m1, m2 is MEASURE
define inheritance m1
    (measure_number = 1 + count(m2 by m1
        where m2 before m1 in measure_in_movement))
```

In this example, m1 refers to a given measure, and m2 is used to count the set of measures previous to m1. The first measure in a movement will have measure number 1, the second will have 2, and so on. This syntax is obviously rather cumbersome. In fact, this example represents a special type of aggregate function (related to the "count" function in this example) that will be discussed in chapter 5. In the course of that discussion a more natural syntax will be discussed.

Upward Inheritance in the Hierarchy: In this case, a parent attribute depends on an aggregate function of the attributes of its children. For example, a beam *group* consists of an ordered set of *chords*. Every group has the attribute *start time* that depends on the start times of its constituent chords: the beam group starts when its first chord starts. *Start time* is thus inherited upward, from chords to groups. This inheritance is defined as follows:

```
define entity GROUP (beam group attributes...)
define entity CHORD (chord attributes...)

define ordering chord_in_group (CHORD) under GROUP

define inheritance GROUP
    (start_time = min(CHORD.start_time by GROUP
        where CHORD under GROUP in chord_in_group))
```

3.3. Implementing Inheritance using Query Modification

When the user presents a query that requires access to the value of an inherited attribute, the system must determine the value of the inheritance expression for that attribute at that point in time. In this section, an implementation of inherited attributes based on query modification is developed. Query modification has been used as a means of supporting both integrity constraints and relational views [Sto75], each of which displays similarities to inherited attributes.

When the system encounters a **define inheritance** statement, it catalogues the association between the (new) inherited attribute names and the entity type of the range variable specified by the statement. The system also catalogues the expression that is associated with the attribute name.

Every time a query references an attribute, the system catalog (the ATTRIBUTE relation) indicates whether the attribute is inherited or not. If it is inherited, the expression is substituted into the query for the attribute value, and the resulting query is then processed.

Here is the algorithm to be performed for every inherited attribute referenced within a query.

Given:

a term of the form $m.n$ within a query Q, where m is a range variable over relation X, and

an inheritance definition of the form:

range of p **is** X
define inheritance p $(y = e)$ **where** q

where e is an expression (possibly involving p), and q is a qualification.

(1) Rename every range variable in e and q other than p so that they don't conflict with range variables used in Q.

(2) Replace every occurrence of p in e by m.

(3) Substitute e for $m.n$ in the query Q.

(4) Replace every occurrence of p in q by m.

(5) Add q to the qualifications of the query Q.

In the following paragraphs, examples are presented for each of the inheritance examples of the previous section.

Find the pitch and start time of every note prior to the note whose start time is less than 100. For this example, assume "pitch" is a native attribute of NOTE. The query is:

```
range of n is NOTE
retrieve (n.pitch, n.start_time) where n.start_time < 100
```

The inheritance statement (from the previous section) is:

```
define inheritance NOTE (start_time = CHORD.start_time)
    where NOTE under CHORD in note_in_chord
```

The result of applying our query modification algorithm to the query is:

```
range of c' is CHORD
range of n is NOTE
retrieve (n.name, c'.start_time)
    where c'.start_time < 100
    and n under c' in note_in_chord
```

The query modification algorithm introduces a unique range variable c' to replace CHORD, as a result of step (1) of the algorithm.

Retrieve the start time of measure number 20. Assume that start time is a native attribute of a measure. The query is:

```
range of m is MEASURE
retrieve (m.start_time) where m.measure_number = 20
```

The inheritance statement (from the previous section) is:

```
range of m1, m2 is MEASURE
define inheritance m1
    (measure_number = 1 + count(m2 by m1
        where m2 before m1 in measure_in_movement))
```

The result of query modification is:

```
range of m is MEASURE
range of m' is MEASURE
retrieve (m.start_time)
    where 1 + count(m' by m
        where m' before m in measure_in_movement) = 20
```

In this example, m' is introduced in step (1) as the unique value for range variable m2.

Find all pairs of groups that begin at the same time. Assume that the "name" attribute is native to groups. The query is:

```
range of g1, g2 is GROUP
retrieve (g1.name, g2.name) where g1.start_time = g2.start_time
```

The inheritance statement (from the previous section) is:

```
define inheritance GROUP
    (start_time = min(CHORD.start_time by GROUP
        where CHORD under GROUP in chord_in_group))
```

The result of query modification is:

```
range of g1, g2 is GROUP
range of c', c'' is CHORD
retrieve (g1.name, g2.name)
    where min(c'.start_time by g1
            where c' under g1 in chord_in_group) =
        min(c''.start_time by g2
            where c'' under g2 in chord_in_group)
```

In this example, the query modification process introduces two unique range variables, c' and c'', for the two references to the inherited attribute, "start_time."

The above examples demonstrate cases where fairly simple queries are modified into comparatively complicated queries involving multiple aggregate functions and so forth. Inherited attributes that involve aggregate calculations are quite common in the musical domain, and techniques to optimize their performance will be explored in chapter 5.

3.4. An Example of Complex Inheritance

Under certain circumstances, the query language itself is insufficient to support an inheritance function. Complete procedural specifications are then required to determine the manner in which an inherited attribute is to be calculated. The procedures to be used in determining an attribute value may be specified in the form of *rules*, as is done in languages such as PROLOG [STZ84]. One attribute

that requires this type of rule-based inheritance is that of the *volume* of a note. This section presents the rules for determining note volume.

Such a rule set would be used, for instance, by a system which reads the music database for the purpose of performing a score (either for analytical reasons, proofreading purposes, or for actual performance).

We begin by determining how volume (in music, known as *dynamics*[7]) is notated in CMN scores. Dynamic markings in music scores take several forms:

- Global indications of absolute volume at the beginning of a movement.
- Dynamic markings associated with voices.
- Dynamic markings associated with chords (or notes).

Another way to look at these dynamic notations is by observing the nature of their effect on notes. Global markings affect all the notes in a movement, in every part. Markings associated with a voice affect all notes in the voice from the point in the marking until the next marking for that voice. Markings for a single chord affect only that chord, and no subsequent ones.

The effect of a marking may be absolute, that is, it may indicate a particular volume level independent of the preceding context, or it may be relative, dependent on preceding context. For example, the indication *f* (*forte*, loud) is an absolute dynamic indication: the following notes are to be played loudly. The indication *piu f* (*piu forte*, more loud) is a relative indication that the following notes should be played at a fixed volume slightly higher than that of the preceding notes. The indication *crescendo* (get louder) is a relative indication that the following notes should begin at this point to increase their volume over time, smoothly to the next dynamic marking, which usually is an absolute marking.

Some markings, the "momentary" dynamics, affect only the notes in the chord under the marking. Subsequent notes are not affected. They may be relative to previous context, as the *sfz* (*sforzando*, suddenly louder) marking, or may be absolute, as is *fp* (*forte-piano*, loud then soft). Figure 3.9

[7] There is a conflict between the computer and music vocabularies in their use of the word "dynamic". In computer science usage, a *dynamic attribute* is an attribute of an entity computed according to given rules. A *dynamic marking*, on the other hand, is a score annotation which (roughly) directs performance volume, according to musical usage. The meaning in any particular instance should be clear from context.

presents a list of the various dynamic markings.

3.4.1. Entities for Representing Dynamic Markings

Figure 3.10 presents the *dynamic* entity. Each dynamic marking is stored in this relation, and is uniquely identified by its *uid* attribute. The location of the marking is determined by the values of *voice_parent* and *sync_parent* which locate the marking in the score by voice and by time, respectively. The *marking* is the actual text of the dynamic marking, for example, "*ff*" or "*cresc*". A given marking is has one ordinate position under its parent voice, *voice_ordinate*, and one under its parent sync, *sync_ordinate*.

Type	Symbol	Name	Meaning
Absolute			
	ff	fortissimo	very loud
	f	forte	loud
	mf	mezzo-forte	medium loud
	mp	mezzo-piano	medium soft
	p	piano	soft
	pp	pianissimo	very soft
Relative			
	cresc	crescendo	get louder
	dim	diminuendo	get softer
Momentary			
	sfz	sforzato	loud
	fp	forte-piano	loud attack, then immediately soft
Modifiers			
	sempre	always	*sempre ff* "still very loud"
	poco	slightly	*poco f* "almost loud"
	subito	suddenly	*sub. p* "suddenly soft"
	piu	more	*piu f* "louder"

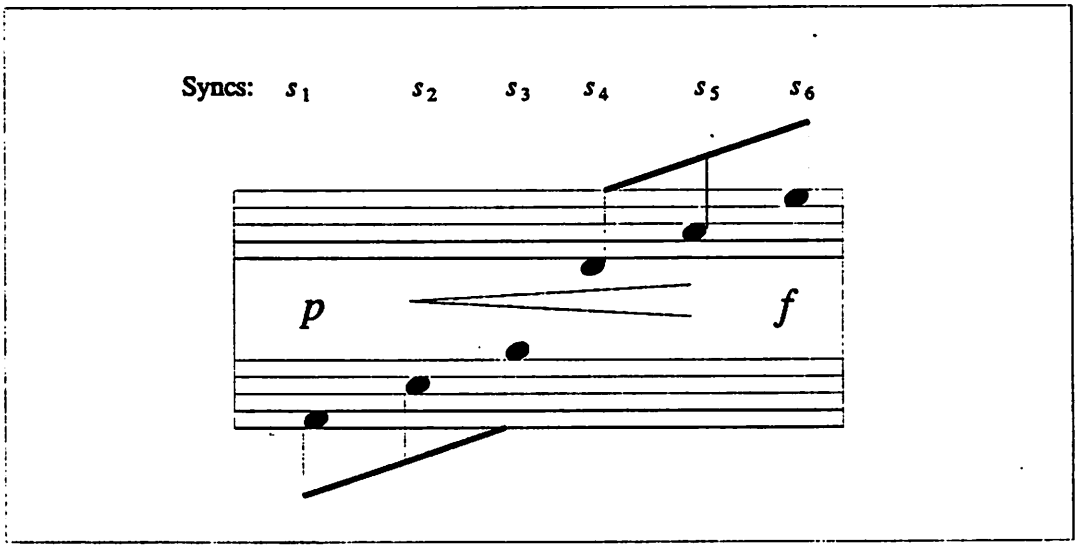
Figure 3.9. CMN Dynamic Markings

DYNAMIC	Native	Inherited
	uid	abs_nearby
	voice_parent	slope
	sync_parent	time
	voice_ordinate	slope_sign
	sync_ordinate	type
	marking	volume

Figure 3.10. The DYNAMIC Entity

There are also inherited attributes in the "dynamic" relation: *abs_nearby* is a boolean attribute which is true if there is an absolute dynamic marking in the same voice in a nearby sync. This is used to fix the value of relative dynamic markings, and will become clear when the actual rules are discussed. *Slope* is the rate of change of volume over time, for notes subsequent to the marking. For a marking such as *crescendo*, it is non-zero. The *time* at which a marking occurs is derived from the time of its sync. This would be expressed in units such as seconds from the beginning of the movement. The *volume*, *slope_sign* and *type* are derived from the mark itself, in the context of various stylistic factors. They indicate the effect of the marking.

Certain dynamic markings which have a linear range, require two entries in the table, the "hairpin" *crescendo* is an example. In figure 3.11, which shows changes in dynamic over a single voice, the volume starts at *piano* in sync s_1, a hairpin *crescendo* begins at s_2 and ends at s_5 so that s_6 is *forte*.



DYNAMICS				
uid	voice	sync	marking	voice ordinate
d_1	v_1	s_1	*p*	0
d_2	v_1	s_2	"start cresc."	1
d_3	v_1	s_5	"end cresc."	2
d_4	v_1	s_6	*f*	3

Figure 3.11. Example of Dynamic Markings

The note entity is just as before, with the addition of inherited attributes for volume, volume slope, and voice. Syncs and chords enter into the calculation, with the attributes already mentioned.

Finally, there needs to be a mapping from dynamic markings to their meanings. This will in general be dependent on the style of the music and the whim of the performer, but figure 3.12 gives a typical example of this DYNAMIC_INTERPRETATION relation.

3.4.2. Database Procedures for Determining Note Volume

Now, for each inherited attribute, a procedure will be given to derive the value of the attribute for a particular entity. For conciseness, the pseudo-code language used for specifying the rules will be functional, in the manner of DAPLEX [Shi81]. This allows the use of inherited or inherent attributes to be syntactically indistinguishable. For example, the uid of a note *n*, an inherent attribute, will be

DYNAMIC INTERPRETATION			
mark	volume	slope_sign	type
f	50	0	ABSOLUTE, PERSISTENT
mf	40	0	ABSOLUTE, PERSISTENT
mp	30	0	ABSOLUTE, PERSISTENT
p	20	0	ABSOLUTE, PERSISTENT
piu f	10	0	RELATIVE, PERSISTENT
piu p	-10	0	RELATIVE, PERSISTENT
cresc.	0	1	ABSOLUTE, PERSISTENT, STARTLINEAR
dim.	0	-1	ABSOLUTE, PERSISTENT, STARTLINEAR
end cresc.	10	0	RELATIVE, PERSISTENT, ENDLINEAR
sfz	20	-1	RELATIVE, MOMENTARY
fp	50	-1	ABSOLUTE, MOMENTARY

Figure 3.12. Dynamic Interpretation Values

represented as *uid* (*n*), and the volume of note *n*, an inherited attribute, will be *volume* (*n*). If an attribute is a uid field, this functional operation can be applied recursively, implying a relational join. For example, "slope(prev(d))" refers to the slope of the dynamic marking previous to the dynamic marking *d*. Occasionally, relational constructs will be required; the syntax for these will be taken from QUEL.

Here are the rules for determining the volume of a note:

```
/* Find the dynamic which "covers" note n */

range of n is NOTE
range of d is DYNAMIC
retrieve d.uid
   where voice(d) = voice(n)
   and time(d) ≤ time(n) < time(next(d))

dynamic(n) ← d.uid

/* Find the volume of the note n */

volume(n) ← volume(dynamic(n)) +
   slope(n) * (time(n) - time(dynamic(n)));

/* Find the slope of this volume for n: */

slope(note) ← slope(dynamic(n));
```

Notice that the retrieve statement is guaranteed to return a single dynamic record, because the time attributes are monotonically increasing. We determine both the volume and the slope of the volume for a note with these rules.

A note also has volume slope, if the volume is to change over the course of the note (this is actually a simplification, since it assumes that the change must occur linearly over the course of the note. For other types of volume change, several notes would have to be tied to together to form a piecewise linear approximation of the change in volume over time).

```
if slope_sign = 0
   then slope(d) ← 0.
   done.

if ENDLINEAR ∈ type(d) then
   if (abs_nearby(d))
      slope(d) ← slope(prev(d))
   else
      slope(d) ← 0;
   done
```

else if (not ENDLINEAR ∈ type(next(d)))

$$\text{slope(d)} \leftarrow \frac{\text{(volume(next(d)) – volume(d))}}{\text{duration(d)}}$$

else if (abs_nearby(next(d)))

$$\text{slope(d)} \leftarrow \frac{\text{(volume(next(next(d))) – volume(d))}}{\text{(duration(d) + duration(next(d)))}}$$

/* untagged end with no nearby fixed */
else

$$\text{slope(d)} \leftarrow \frac{\text{(slope\_sign(d) * DEFAULT\_CRESC)}}{\text{duration(d)}};$$

A few characteristics of this type of rule set are worth noting:

It is essentially expressible in a relational language such as QUEL, with the proviso that control structures (e.g. if-then-else constructs, in this case) must be made available.

The *prev* (for *previous*) and *next* operations are defined on orderings, which must therefore be supported. The result of these operations is a *uid*, from which attributes may be projected. This corresponds to a relational join. Other operators, for example, *chord(note)*, when composed in this way, also imply a relational join. In this example, the join is between the chord.uid and note.chord_parent in the chord and note relations.

The set of entities on which a given note volume depends is easily determined by running the rule set to completion, and taking note of every object in the database that is read. The database system, by a locking mechanism as developed in [SAH85], may cache the resultant values in the database, and only recalculate them when relevant entities are updated. This is known as early evaluation. Alternatively, the data manager may invoke the database procedure at the time the inherited attribute value is requested by the client. This is termed lazy evaluation.

In fact, the data manager may materialize inherited attribute values at any time between operand update and value retrieval. In particular, a database daemon may perform this materialization asynchronously to client access. When a large set of values need to be materialized, the optimal strategy is to materialize first those that will be needed soonest by the client. When this information is not known, heuristics similar to those used for existing buffer prefetch strategies may be used.

3.5. Summary

In this chapter, we have presented the semantics of hierarchical ordering in detail. They are represented schematically by HO graphs at the schema level, and by instance graphs at the entity instance level. The power of this construct lies in its ability to model the "part of" relationship, where an entity consists of an ordered set of other entities.

This construct is found throughout the schema for musical information. The instance graphs for music are very complex, with a large number of subordinate objects in even conceptually small amounts of music, such as a single bar.

An advantage of HO graphs lies in their semantic power in organizing attribute inheritance. We have shown three types of inheritance: upward along the hierarchy, downward along the hierarchy, and laterally over ordered aggregations. All of these inheritances may be mapped into a query language such as QUEL, with suitable extensions to the data definition language.

We have demonstrated a form of query modification that can be used to support this inheritance, by translating references to inherited attributes into expressions containing references to native attributes.

Certain types of inheritance require a more general specification mechanism, and we explore one example of such an inherited attribute in detail. In this example, a general procedural specification is used to define the relationship between notes and their performance volumes.

Throughout this chapter, we have developed a data definition language to represent entities, relationships, and the hierarchical orderings in which they participate within the music database. This language may be used to represent the data model itself, in the form of a meta-database.

In the next chapter we provide a complete description of the entities of the musical notation database, and the hierarchical orderings in which they participate.

CHAPTER 4

A Database Schema for Common Musical Notation

In order to allow a user to refer to meaningful units of musical information, it must first be determined what those units are. This chapter analyzes in detail the entities that compose Common Music Notation (CMN), and their interrelationships.

Section 4.1 begins with an overview of the entities of the CMN schema, and categorizes them into several different *aspects*. An HO graph will be presented for each aspect of the CMN score.

Section 4.2 focuses on the particular aspect of musical notation that represents temporal information. This presentation serves as an example of an application of the data modeling techniques introduced in the previous chapter. Although the representation of temporal attributes will be covered in detail, the discussion is intended to be accessible to those with little background in musical notation.

Section 4.3 continues the exercise of the previous section, developing the HO graphs for the remaining aspects. In focusing on the details of CMN, this section assumes familiarity with musical notation on the part of the reader.

Section 4.4 takes a very small fragment from a musical score, and, using the HO graphs already developed, presents the instance graph for this specific musical example.

Finally, in section 4.5, some published scores are analyzed, and the approximate size of the databases we might expect to build from them is determined. A simple predictive model is proposed for determining the size of a database representation based on the information density of the score.

4.1. CMN Entities

In many data management domains, there are only a handful of entities. For example, the standard company database contains employees, jobs, departments, parts, suppliers, and orders. Musical information has, even at first glance, many more entities than this. These entities are summarized in figure 4.1, and will be discussed in the following sections.

Entity type	Description
Score	The unit of musical composition.
Movement	A temporal subsection of the score.
Measure	A temporal subsection of the movement.
Sync	Sets of simultaneous events.
Group	A group of contiguous chords and rests in a voice.
Chord	A set of notes in one voice at one sync.
Event	An atomic unit of sound; one or more notes.
Note	An atomic unit of music; a pitch in a chord.
Rest	A "chord" containing no notes.
MIDI	A MIDI note event.
MIDI control	A MIDI control event at a point in time.
Orchestra	A Set of Instruments performing a Score
Section	A family of instruments.
Instrument	The unit of timbral definition.
Part	Music assigned to an individual performer.
Voice	The unit of homophony.
Text	In vocal music, a line of text associated with the notes.
Syllable	The piece of text associated with a single note.
Page	One graphical page of the score.
System	One line of the score on a page.
Staff	A division of the system, associated with an instrument.
Degree	A division of the staff (line and space).
Graphical Definitions	All the graphical icons and linears.
Instrument Definitions	Instrument patches and specifications.
Other graphical attributes	Accents, Accidentals, Annotations, Arpeggii, Barlines, Beams, Clefs, Duration dots, Fingerings, Flags, Hairpins, Key signatures, Meter signatures, Note heads, Rests, Slurs, Staff lines, Stems, Ties, Letters, etc.

Figure 4.1. The Entities of a CMN Schema

4.1.1. Aspects of CMN

Each musical entity contains various attributes. For example, attributes of a "note" entity are its position, shape, size, start time, parent chord, and so on. Musical entities in the CMN score have several aspects and subaspects, as shown in figure 4.2. These may be thought of as different views on the musical schema.[1] Roughly, the temporal aspect pertains to *when* musical events are performed. The timbral aspect refers to *how* they are performed (e.g. by what instrument, at what pitch, how loudly, etc.). This aspect itself admits a finer characterization, into pitch, articulation, and dynamic (i.e. volume) subaspects of the data. The graphical aspect is concerned with how musical events are notated graphically. A subaspect within the graphical aspect of the score is concerned with with tex-

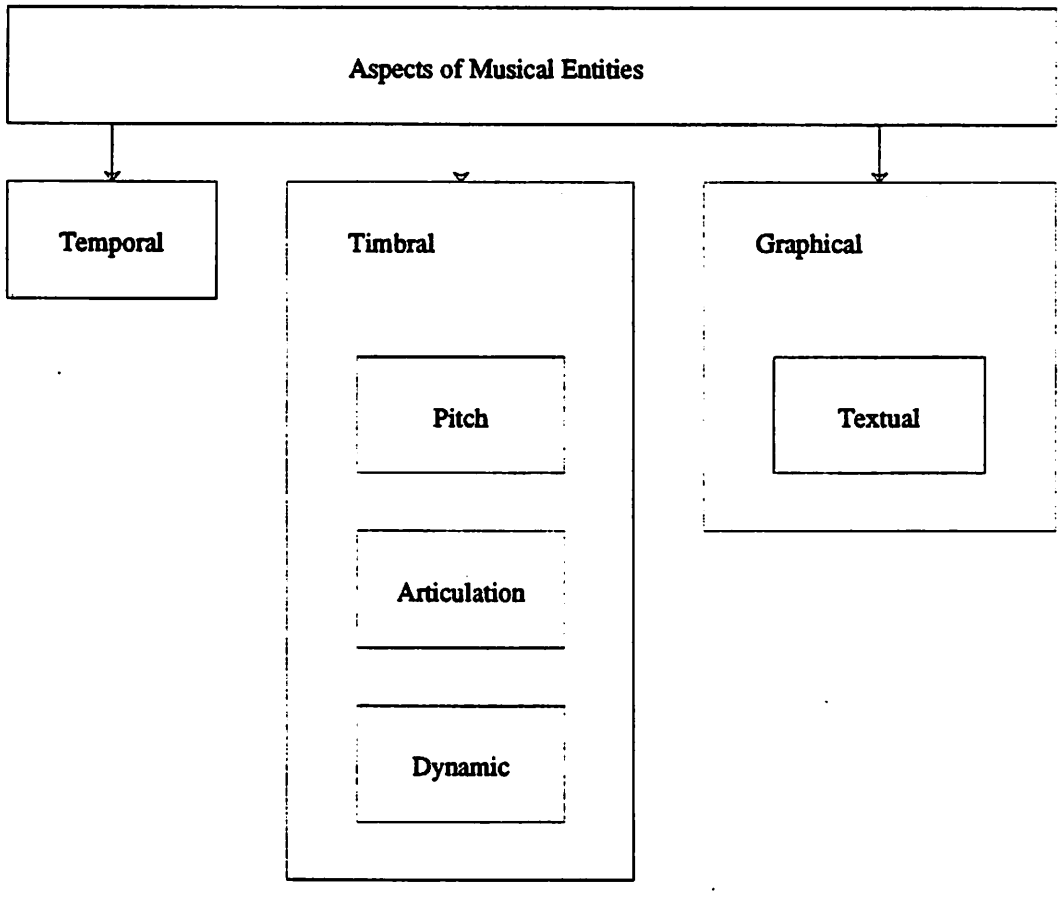


Figure 4.2. Aspects of Musical Entities

[1] The term *view* has a specific technical meaning in databases, thus the term *aspect* will be preferred in this discussion.

tual material, including a variety of score annotations, as well as the lyrics (or *libretti*) associated with melodies.

The utility of this notion of aspect may be suggested by example. A musical note, as it appears on a score page, possess attributes associated with each of these aspects.

The temporal aspect of a musical entity refers to those attributes and relationships which model the entity's placement in time. A note has attributes related to the time at which it is performed in the course of a composition.

Because CMN groups musical events by "instrument," one may refer to the timbral aspect of certain entities. A note has a timbral aspect that refers (roughly) to the instrument that "performs" it.

A note may have several attributes reflecting its pitch aspect. These include such things as its staff degree, associated accidentals, and relations to key signatures and clefs. There is also a notion of performance pitch (either MIDI key codes or frequency information) that is indirectly associated with notes.

A note inherits various articulative attributes. These reflect roughly how the note is performed. They include modal attributes such as *staccato* (shortened or clipped) or *marcato* (marked or stressed). Also, a note may have inherited various performance attributes, such as when a violin note is played *pizzicato* (plucked) or *arco* (bowed).

Another attribute which a note must inherit is its dynamic value, which indicates how loudly it is to be played. In the graphical score, these are given as annotations such as *forte* (loud) or *pianissimo* (very soft). Such attributes are not typically assigned directly to a note, but rather are inherited by the note from the context in which it lies.

Finally, since CMN is a graphical notation, musical entities have a graphical aspect, relating to their representation on the written page. For a note, this includes its various graphical components, such as the note head, stem, associated accidentals, flags, dots, accents, and so on. Each of these has a shape or size and location on the page. These are all graphical attributes. A subclass of graphical objects on the score page may be considered to be textual objects. Although individual note entities do not have a textual aspect, there are a variety of textual annotations associated with pages, systems, staves, syncs and individual chords.

4.1.2. Hierarchical Ordering Graphs for CMN Aspects

Two strategies are used here to organize the representation of musical entities. First, the entities are arranged into groups by the aspects in which their attributes participate. Not every entity has attributes in every aspect (MIDI events, for example, have no graphical aspect in CMN). Many entities, as was seen for note entities, will appear in the graphs for several aspects. For each aspect, an HO graph is defined. Towards the top of each graph will be abstract structures that give form to the music. At the bottom of the graph will be the low level objects that make up the physical attributes of the music.*

4.2. The Temporal Aspect

Before discussing the entities involved in the temporal aspect of a CMN score, certain uses of the word "time," as it appears in music, must be defined. Specifically, a distinction must be made between "performance time" and "score time."

The location in time at which a musical event is actually initiated, and how long it lasts, are recorded in performance time. The units of performance time are seconds. Score time, on the other hand, is measured in rhythmic units. Musical structures in CMN, such as notes, chords and measures, may fall into a more or less regular rhythmic structure whose unit is the *beat*.

The duration of a beat, however, is consistently distorted in performance. This distortion may be noted in the score, by directives such as *accelerando* to speed up a passage or *ritardando* to slow down. Alternatively, they may be inherent in the style of the music, as in the *rubato* associated with certain musical styles. Thus the mapping between the location of events in score time, and their location in performance time, may be arbitrarily complex. When an orchestra performs, it is the role of the conductor to establish this relationship between score time and performance time.

The HO graph for temporal attributes may now be considered. The relationships among the temporal aspects of musical entities are shown in figure 4.3. To review the elements of the graph, each box contains one or more entity types. The solid arrows refer to hierarchical ordering of child types under a parent type, while the dotted arrows indicate hierarchical ordering under entities not shown in this graph (they appear under other aspects). The indirect relationship indicated by the dotted lines arises because the given HO graph does not include all the entities in the musical schema.

* In this discussion, entity names will appear in *italics*.

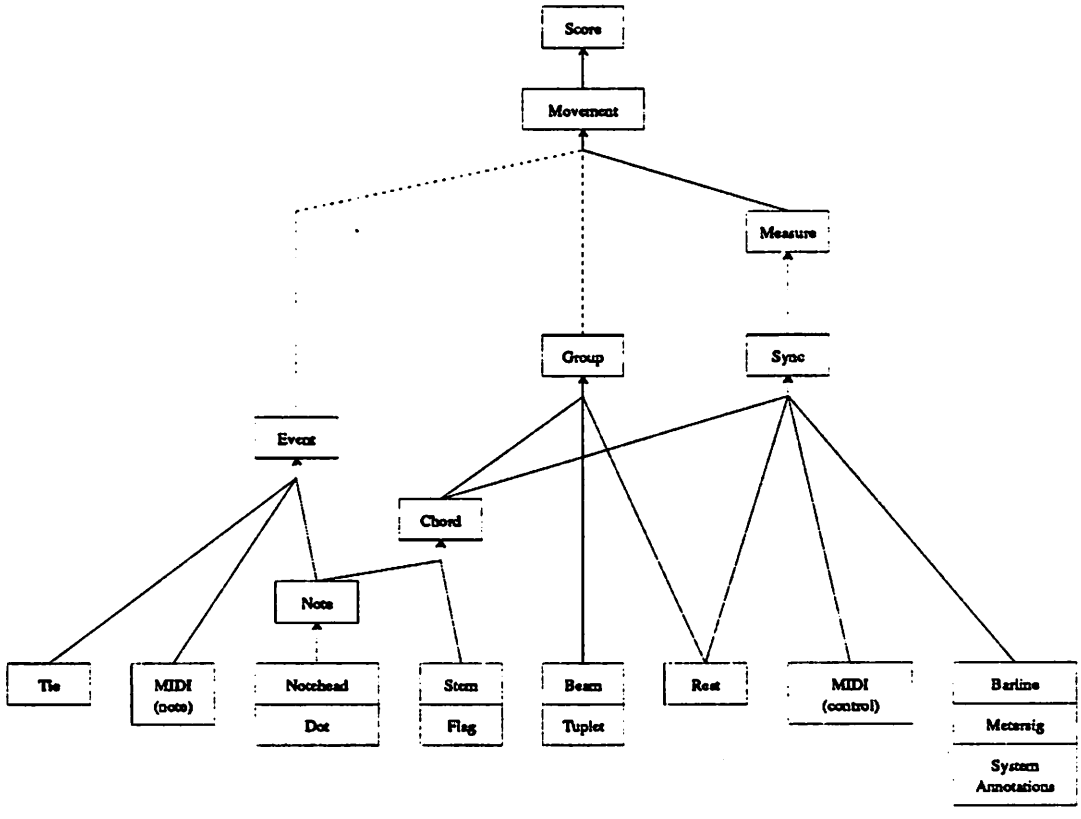


Figure 4.3. Temporal Relationships in the CMN Schema

A musical *score* is the compositional unit of the database. Its temporal attribute is the duration of the composition. This duration is the sum of the durations of its constituent *movements*. A movement is a somewhat arbitrary (though widely used) unit of performance. These movements are further subdivided in time, into *measures*. Measures determine rhythmic divisions of a passage. Where a musical passage has a rhythmic pulse (i.e. a *beat*), each measure consists of an integral number of such pulses.

The various musical events within a passage (such as notes) are typically aligned on these pulses. Each such point of alignment constitutes a *sync*. This term is taken from the Mockingbird system [MaO83]. A sync has, as a temporal attribute, the point in score time at which it occurs. This can be specified as a number of beats (units of score time) from the start of the measure in which the sync occurs. Figure 4.4 shows how a measure is divided into syncs. The notes within a sync are grouped

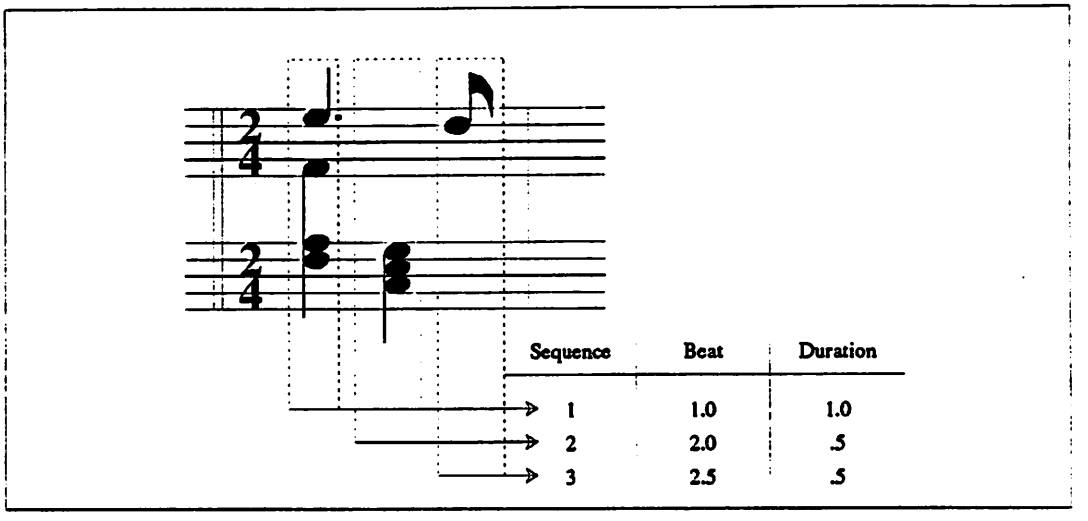


Figure 4.4. Dividing a Score into Syncs

into *chords* (by voice, as shall be shown in the timbral definition). The start times of notes and chords are inherited from their parent syncs.

In addition to the grouping of chords into syncs into measures, particular musical voices may be independently organized into melodic *groups*. Groups have a variety of semantic functions in music. As shown in figure 4.5, these include phrasing (e.g. notes covered by a slur) and timing (e.g. beams and tuplets). A group has a the temporal attribute, "duration," which is a function of the duration of its constituent *chords* and *rests*.

Rests, like chords, have temporal location and duration, although they result in no performance (MIDI) information.

An *event*, from the temporal point of view, determines the placement in time of each atomic unit of sound. It has a unique start and end time, and is performed by a specific voice. An event is thus a unit of performance. A *note*, on the other hand, is the notated unit of music. These two are not necessarily the same, as, for example, when two notes are tied together. The *Tie* is a musical construct that binds multiple note entities under a single event entity.

At the bottom of the graph appears the *MIDI* entity. This assumes a MIDI model [Jun83], where individual musical "events" have particular starting and ending times. For scores that use CMusic

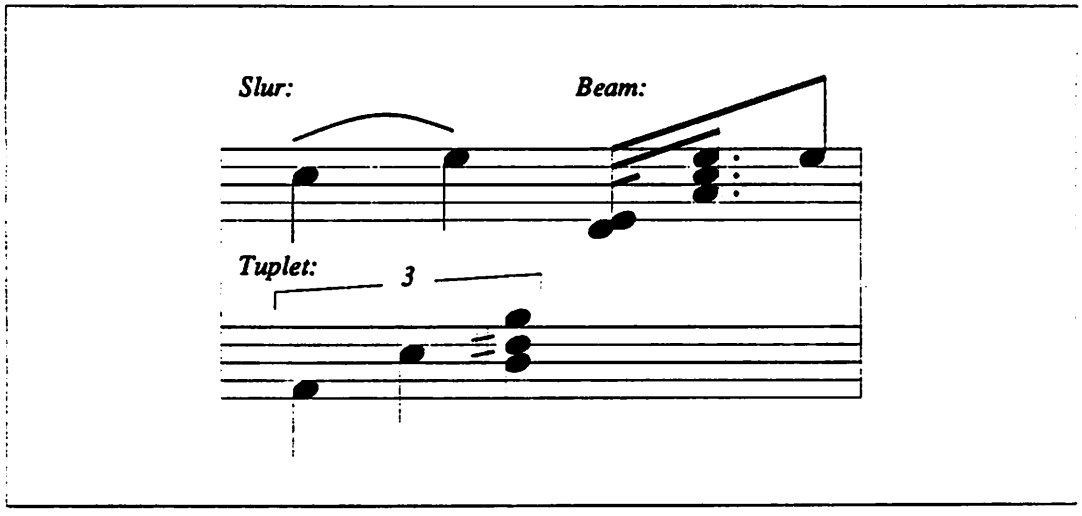


Figure 4.5. Examples of Chord Groups

style note lists, these can easily be extrapolated from the MIDI event information. MIDI events constitute performance information, and so their temporal parameters are given in performance time (i.e. seconds). There are MIDI commands to control note events, as well as control information such as the actuation of a control switch other than a keyboard key (e.g. the *sostenuto* pedal of a piano).

4.3. Other Aspects

For completeness, the HO-graphs for the remaining aspects of CMN are presented here. The description of these graphs will be more terse than that of the previous section, and familiarity with CMN is assumed.

4.3.1. The Timbral Aspect

We now consider those entities that have attributes which relate to the type of sound they model, that is, their timbre. The HO graph for the timbral aspect of CMN is shown in figure 4.6.

Apart from the description of individual scores, the database contains *definitions* for each kind of instrument. For the classical composer, an instrument has various attributes such as its family (e.g. the trombone belongs to the brass family), pitch range, notational transposition, standard clef, and so on. For the composer of synthesized music, an instrument may be defined by a "patch," the set of

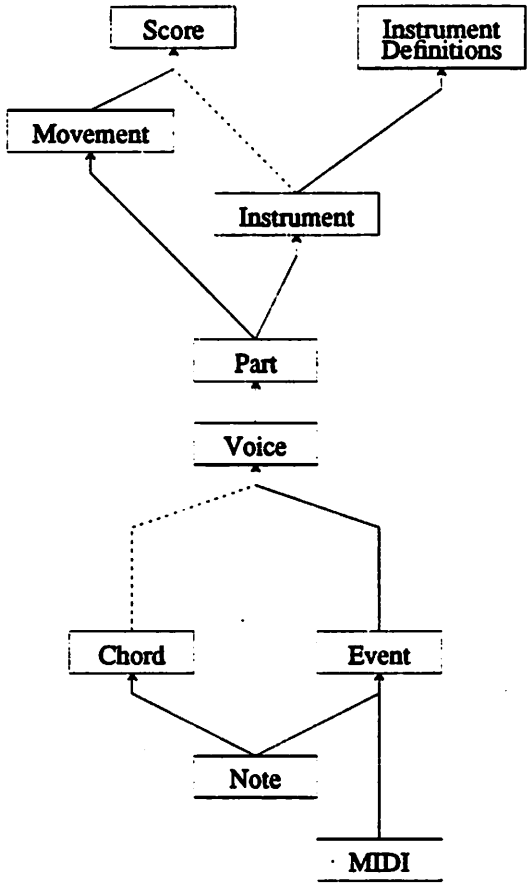


Figure 4.6. Timbral Relationships in the CMN Schema

parameters for a given piece of sound synthesis equipment, or by an algorithmic definition of the sound, such as a CMusic instrument definition.

The *instruments* used within a composition refer directly to these definitions, and indirectly (via graphical constructs to be described momentarily) to the individual *score*. The score itself is divided into *movements*, as already mentioned. In this schema, movements have not only a temporal aspect (as shown in the previous section), but a timbral one as well, in that each timbral voice of the score is defined (arbitrarily) to be one instance of an instrument over the course of one movement.

A *part* is a single instance of an instrument. For example, in a symphony, there is a single musical instrument known as the "violin." A composition may be scored for several violin parts, typically named "first violin," "second violin," and so on. When modeling compositions for acoustical

instruments, each part represents a single instrument (or set of instruments performing strictly in unison). For compositions destined for synthesized voices, the parts are typically associated with individual MIDI channels.

Each part consists of one or more *voices*. Musicologically, a voice is defined to be a homophonic subset of a part. In other words, while a voice may contain multiple simultaneous notes (such as a chord), it does not contain any polyphonic structure. Each chord within a single voice must end (in score time, not necessarily in performance time) before the next chord in that voice may begin. For instruments such as the violin, when played conventionally, each part consists of one voice, since the violin is not played polyphonically. For harp or piano, which have the ability to produce polyphonic textures, a part may consist of many voices. Synthesizers are commonly configured both polyphonically and monophonically, depending on their technical capability and the will of the composer.

Voices consist of non-overlapping sequences of chords which in turn consist of sets of simultaneous notes. These entities, groups, chords, notes, as well as events and MIDI information, all inherit their timbral attributes from the voice entity of which they are a part. For example, the MIDI channel associated with a particular instance of a MIDI entity is inherited from the part, containing the voice, containing the event, containing the MIDI command.

4.3.2. The Pitch Aspect

Associated with many (not all) events is a notion of pitch. Some instruments, such as cymbals and certain drums, have no pitch associated with their events. Figure 4.7 shows the ordered aggregate graph for the pitch aspect of entities in the CMN schema.

Just as the temporal aspect reflects both score time and performance time, the pitch aspect reflects both notated pitch and performed pitch. When the score is performed, pitch refers roughly to the fundamental frequency of the performed note. This pitch attribute is associated with the *event* entity, and is inherited by *MIDI* objects.

Notated pitch reflects a semantic pitch concept which is not exactly identical to this performance attribute. These notated entities, note heads, accidentals, clefs, and key signatures, are shown in figure 4.8. The staff *degree* (in the context of a given *clef* for that staff) on which a note is placed, and the *accidentals* associated with the event group, determine the pitch with respect to the tonality of the

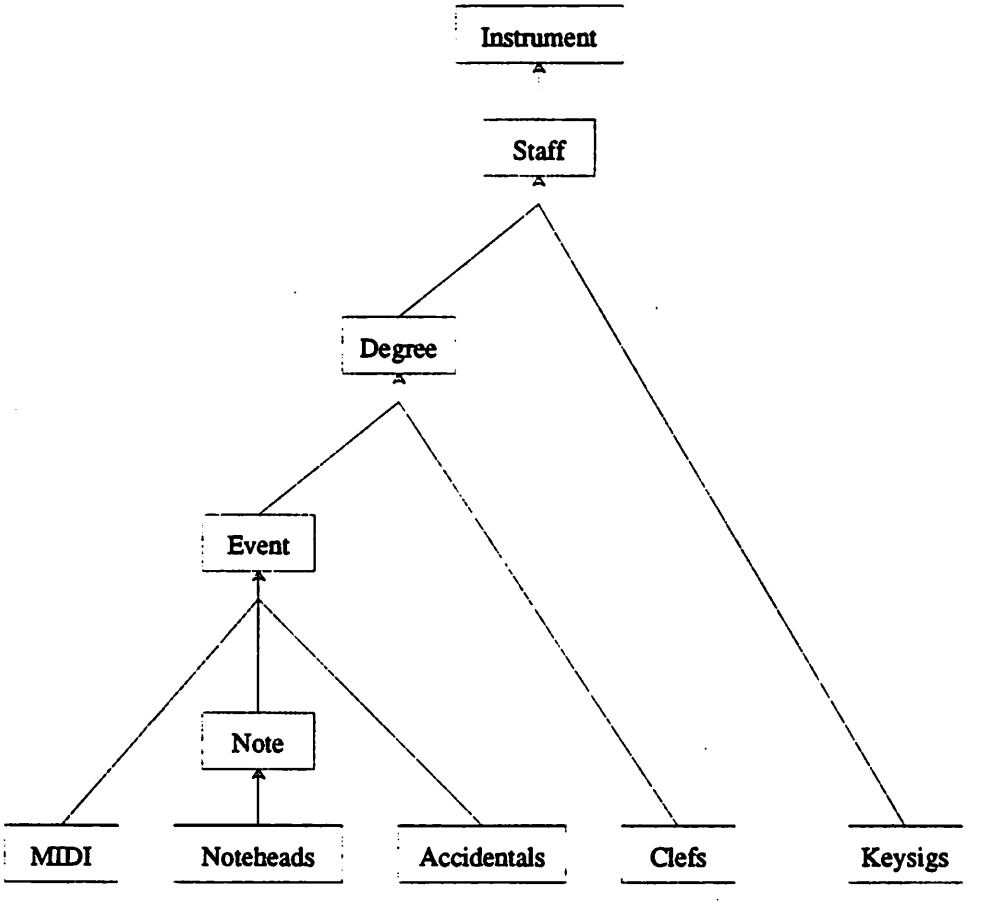


Figure 4.7. Pitch Relationships in the CMN Schema

composition, as indicated by a *keysig* (key signature) entity. Thus there are enharmonic notes: different notated pitches (such as b-natural and c-flat) that refer to equivalent performance pitches (figure 4.8).

4.3.3. The Articulation Aspect

Various aspects of performance nuance are grouped under the category of articulation. The HO graph for entities that have an articulation aspect is shown in figure 4.9.

The ways in which chords are emphasized or accented, pointed or broadened, fall under this category. The entities that mark these attributes of a chord are *accents* or various *annotations* such as *fermate*, or ornaments such as mordents and trills.

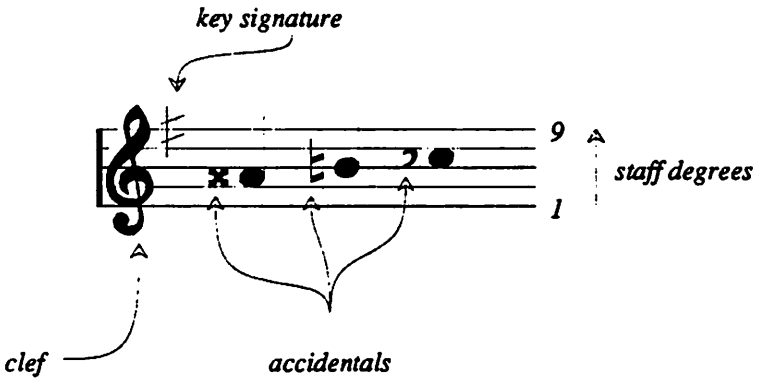


Figure 4.8. Pitch Entities: Enharmonic Pitches

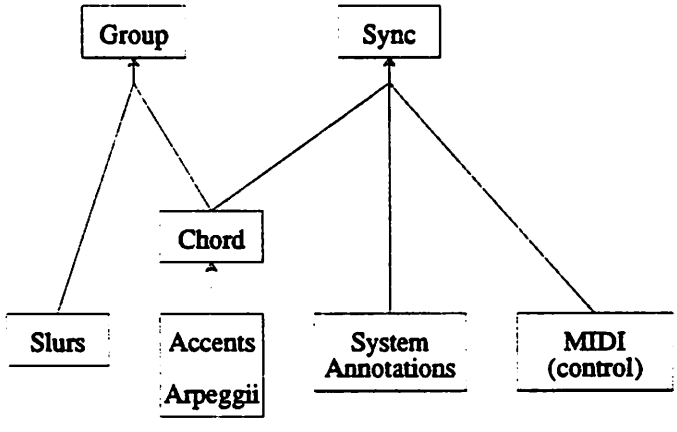


Figure 4.9. Articulation Relationships in the CMN Schema

Groups of chords are also subject to articulation. When applied to groups, this is typically known as phrasing. It is indicated by *slurs*, as well as staff annotations such as *caesurae*. Because of the subtle nature of phrasing, it is often not indicated directly in the score, but implied by other properties, such as dynamic contour, pitch contour, or orchestration. These may be represented by instances such as MIDI control entities, which often serve an articulative function. Although not reflected directly in CMN, they are made available in the database to effect various nuances of performance.

4.3.4. The Dynamics Aspect

Attributes associated with the dynamic aspect model the loudness of musical events.[2] The entities associated with these attributes are shown in figure 4.10.

A score may contain *hairpins* (for lack of widely accepted term) that indicate increasing or decreasing volume in a voice over score time. These are relative dynamic markings. Absolute dynamics are also specified by *dynamics annotations* such as *f* (*forte*) for loud and *p* (*piano*) for soft. There is a fixed class of approximately twenty such annotations (as discussed in section 3.6). Events have a performance dynamic attribute which is derived from these notational cues. MIDI entities inherit the dynamic attribute of their *event* parent.[3]

4.3.5. The Graphical Aspect

Because CMN is a graphical notation intended to capture both the temporal and timbral features of the score, the graphical aspect of the schema is more complex than the previous ones. Figure 4.11 shows the ordered aggregate graph for the graphical aspect of CMN, excluding the textual sub-aspect, which will be considered separately.

In this schema, the *score* and *movement* entities have graphical attributes such as title page information (e.g. title, composer, date of composition, librettist, etc.). A movement is divided into a number of *pages*. A page is divided into one or more *systems*. A system corresponds to one line of music. In orchestral scores, there is often one system per page. For single instrument scores, many

---

[2] This is slightly different than volume, insofar as changes in loudness typically involve timbral modifications in addition to volume change.

[3] Unfortunately, the MIDI protocol has no means to specify continuous change in volume within a single sound event. It may be roughly simulated by concatenating several MIDI sound events within an event entity, each with a stepwise change in volume.

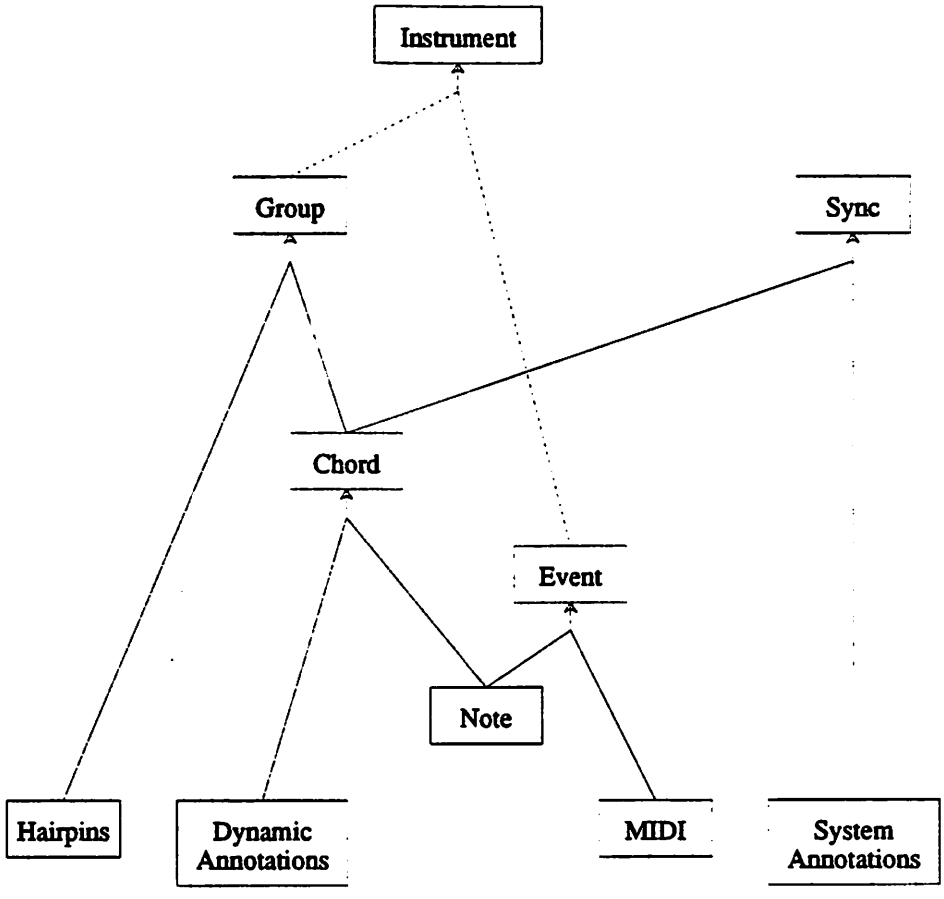


Figure 4.10. Dynamics Relationships in the CMN Schema

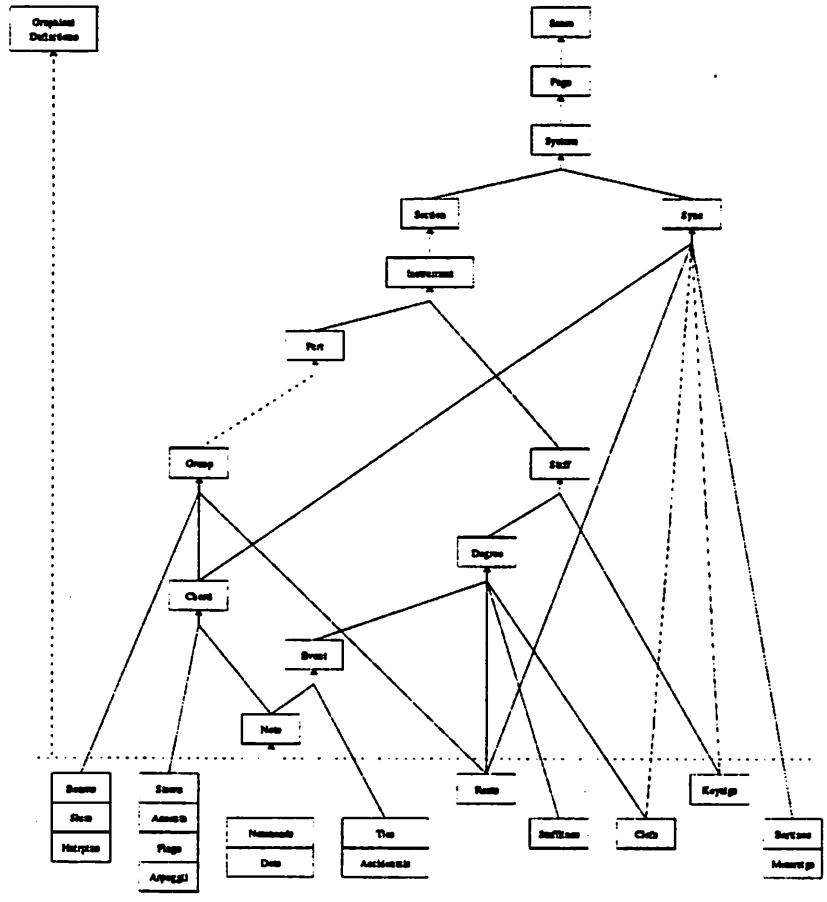


Figure 4.11. Graphical Relationships in the CMN Schema

systems per page are possible.

A system is a two dimensional object. Along the horizontal dimension, it is divided into syncs. The temporal aspect of a sync (e.g. the point in time at which the sync begins, and its duration) have already been discussed. The graphical aspect of a sync is reflected in its position on the page: a particular x-coordinate within the system, around which graphical components of the sync are built. Although those graphical objects (note heads, stems, accents and so on) may not lie directly on the graphical sync position, they all are placed with reference to it. If a graphical sync moves (for example, as the result of an editing operation), all the constituent graphical objects must also move.

The vertical dimension of the system is divided hierarchically. First, the system is separated into orchestral *sections*. All instruments in the same musical family are thus grouped together. Each sec-

tion is divided into *instruments*, each of which consists of one or more *parts*. Independently, associated with each instrument are one or more *staves*. Figure 4.12 shows graphically how this hierarchy is represented in the orchestral score. This figure shows an example of multiple parts (Violin I and Violin II) on one staff, and multiple staves in one part (Piano). In CMN, sections that cover more than one instrument are bounded by a square bracket, and multiple staves in a single instrument are bounded by a curly bracket.

Each staff consists of five lines, each line and each space between two lines constitutes a staff degree. Graphically, a *degree* is half the vertical distance between two staff lines. This vertical dimension, within the staff, partly determines the pitch of *events* on that staff.

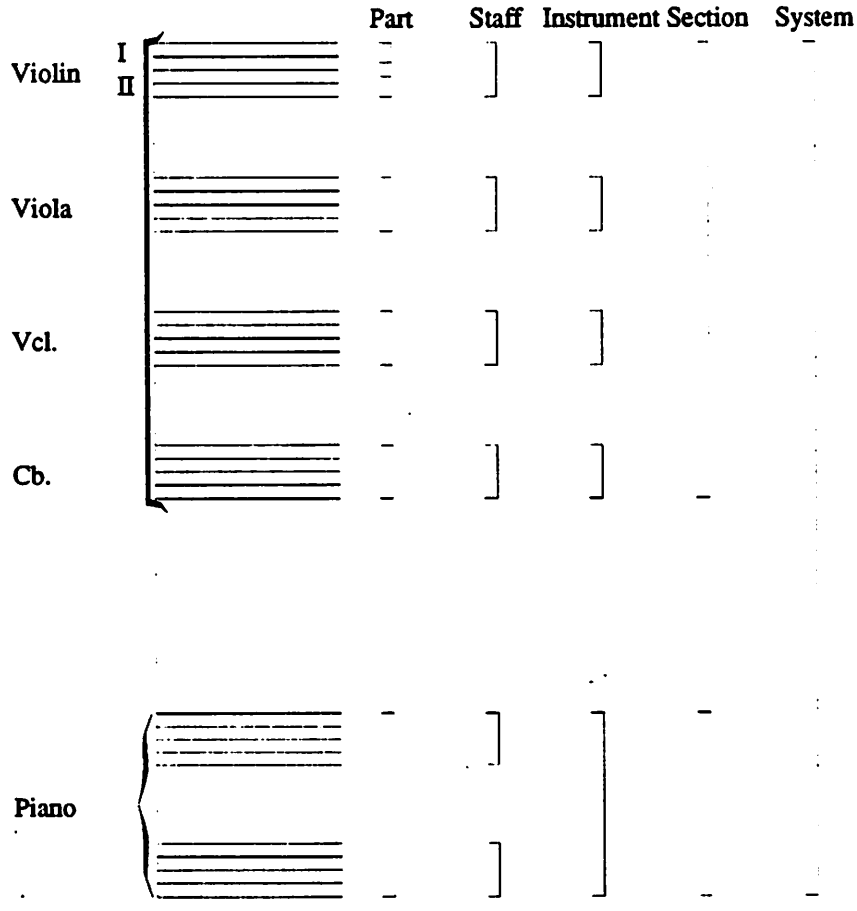


Figure 4.12. A Musical System

A rather large collection of graphical objects are associated with these various structures. Some of these are shown in figure 4.13. All of them have *definitions* that define their graphical structure in some graphics representation language (such as Postscript [Ado85]).

Most of these graphical objects have already been described under other aspects of the model. The few that remain provide additional information to the score reader on interpretation of the score. For example, *staff lines* provide the score reader with a reference grid to easily determine the degree of other graphical entities.

4.3.6. The Textual Aspect

Other aids take the form of arbitrary *annotations* which provide terse textual comments on the score or its performance. The annotations are applied in stylized fashion to various other entities. Thus there are page annotations associated with the page entity, part annotations associated with the

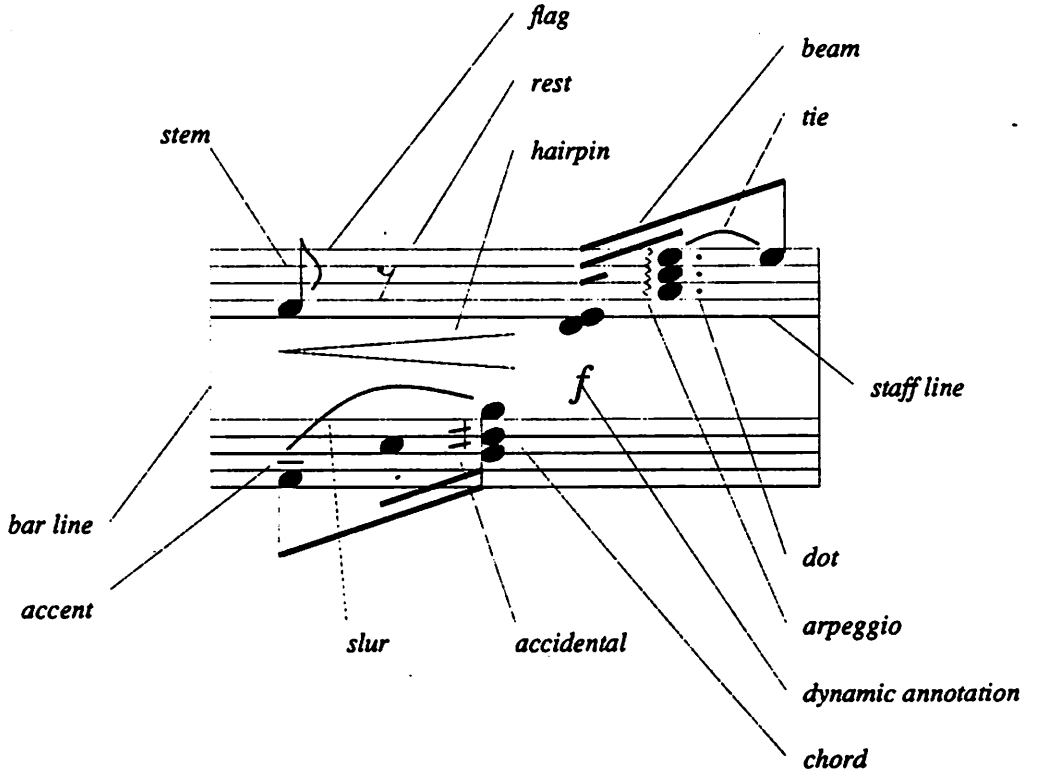


Figure 4.13. Graphical Entities

part entity and so on. Figure 4.14 shows the ordered aggregate graph for the textual aspect of CMN.

Score and movement annotations include textual information relating to the title of the composition, and other textual material that comes at the head of the piece. Instrument and part annotations include text that labels the left margin of the system, indicating, for example, which instrument plays the music on a particular set of staves.

System annotations occur at a particular sync, and usually are notated above the system. They include textual annotations that give performance directions that apply to all parts at a point in time (e.g. tempo indications such as *Allegro*, "quickly").

Various other indications annotate a particular staff at a point in time. These staff annotations include information on how the score should be read (e.g. "*a due*," where one voice is to be read by two parts sharing a staff), or how it should be performed (e.g. "*pizzicato*," where the notes on a staff should be plucked rather than bowed by a stringed instrument).

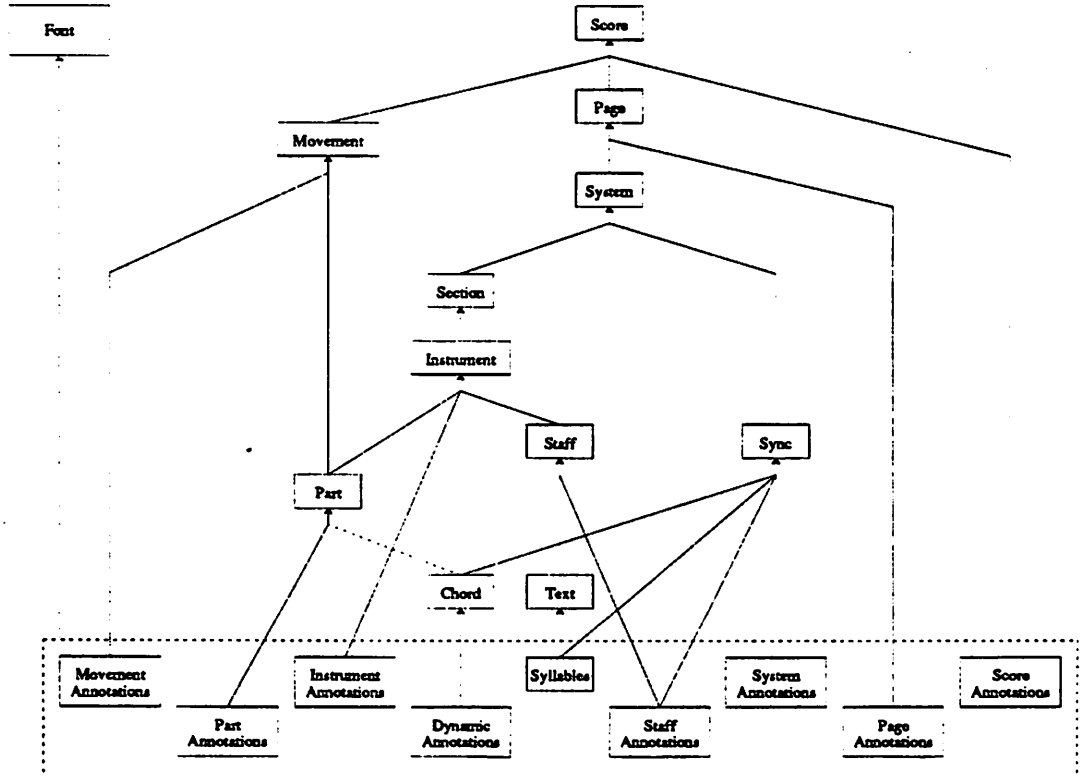


Figure 4.14. Textual Relationships in the CMN Schema

Dynamic annotations are associated with a given chord to indicate that this chord, and perhaps subsequent ones in its voice, should have a particular dynamic level. These include annotations such as "*f*" for *forte* (loud) and "*p*" for *piano* (soft).

In vocal music, where words are spoken at particular pitches, the words themselves are represented textually in the score. In this case, lines of *text* are aggregated under each staff. The text is divided into *syllables*, each under a single sync. This term is used in a somewhat stylized fashion. These vocal "syllables" are not necessary the actual syllables of a word, but rather those word parts notated a one point in the score (multiple syllables are often sung on one note).

Every instance of a textual annotation has a particular font associated with it. Sometimes the font is fixed by convention, as with the characters used in piano fingerings or dynamic markings. Other times the notator has considerable latitude in selecting fonts. For example, the lyrics of a song might be in Roman or Italic characters, and the title of a composition might be set in a variety of typefaces or sizes.

This concludes the description of the entities in the musical database schema. A prototype implementation of this schema has been made, and is included in appendix C. By way of summary, this prototype contains 55 entities, with 251 attributes, both native and inherited.

4.4. An Example from Music

In this section an instance graph for a small musical example will be developed. Figure 4.15 shows a fragment of music, representing one measure from a piano score. For reference, each of the chords in this figure is indicated by a dashed box. In spite of the small size of the fragment, it contains large number of entities: a measure, a part, syncs (sets of simultaneous chords), voices, chords, staves, notes, and graphical elements such as flags, stems, accents and dots. In figure 4.16, musical icons have been replaced by database entities, each represented by a named box (for the purposes of the example, the set of entities has been simplified slightly). The entities are positioned in figure 4.16 so as to roughly correspond with their actual locations in the measure of music as shown in figure 4.15. The four chords are again indicated by dashed boxes.

These entities form the nodes of the instance graph. To determine the P-edges and S-edges of the instance graph, refer to the HO graph for this set of musical objects, shown in figure 4.17. Again,

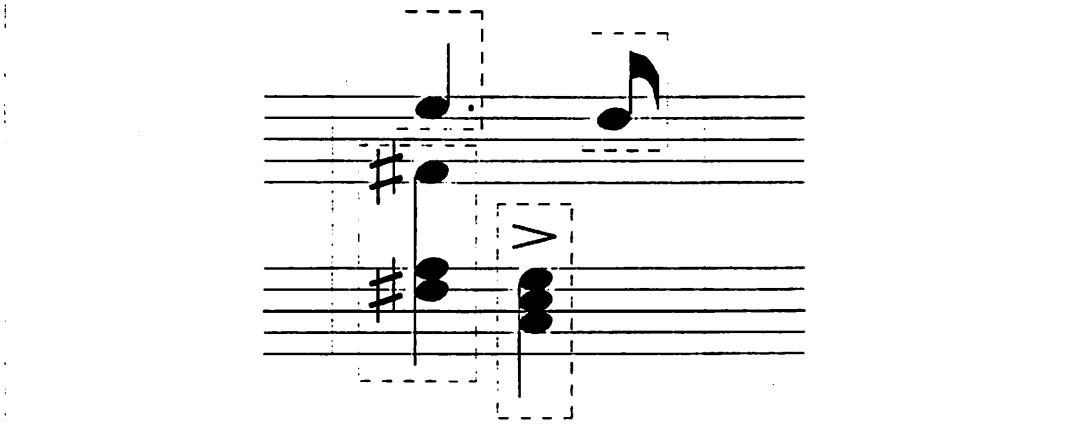


Figure 4.15. A Measure of Music (chords are indicated by boxes)

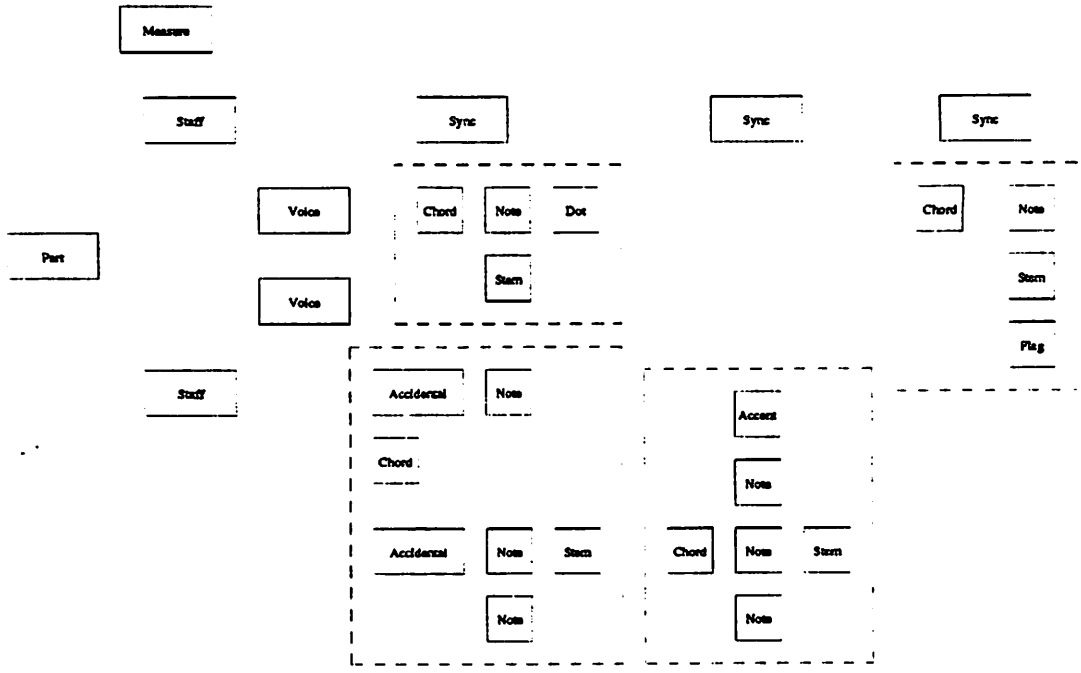


Figure 4.16. Entities of the Instance Graph

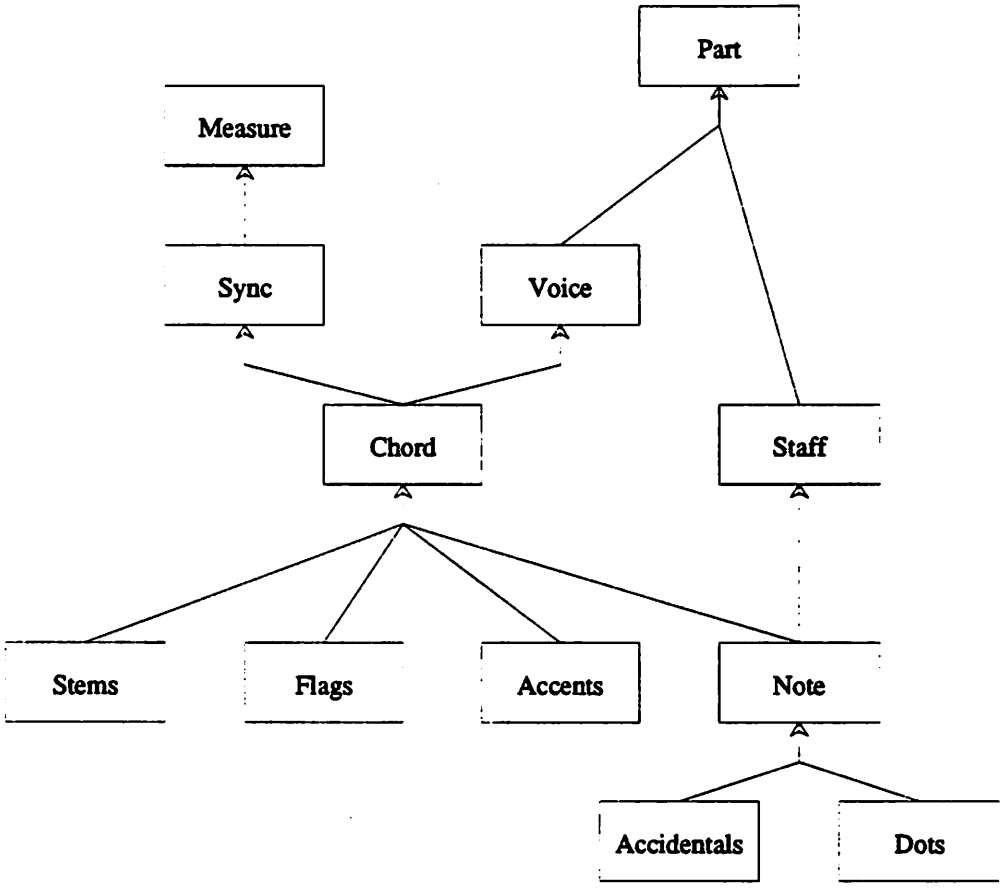


Figure 4.17. HO Graph for a Subset of Musical Entities

each edge in this HO graph defines one ordering on the musical data.

Unfortunately, if one were to view all the edges of the instance graph simultaneously, the graph would be unreadable. Therefore, the next five figures each show a subset of the edges in the instance graph for our example.

Figure 4.18 shows the ordering of the staves and voices of this measure under the *part* entity. Figure 4.19 displays both the ordering of syncs within measures and the ordering of chords within each sync. The ordering of chords within syncs is shown in figure 4.20, along with the orderings of stems, flags and accents under chords. Notice that there are no S-edges for this latter group of orderings, since each such ordered set in our example contains a single stem, flag, or accent. This is not necessarily true. For instance, there could be multiple flags associated with a given chord.

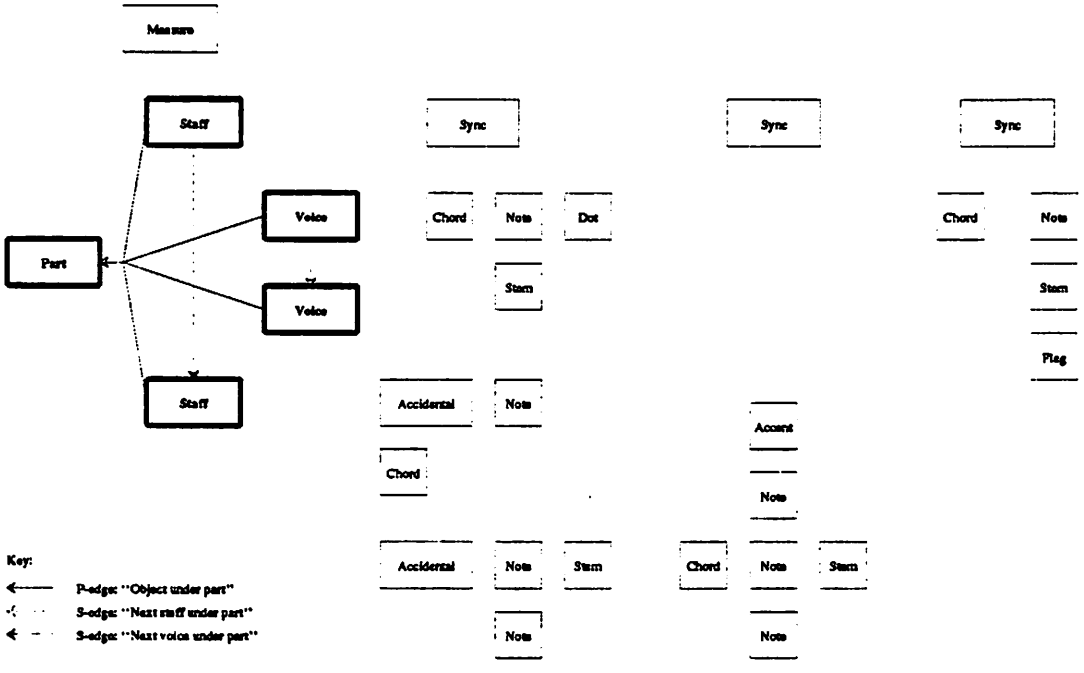


Figure 4.18. Orderings under the "Part" Entity

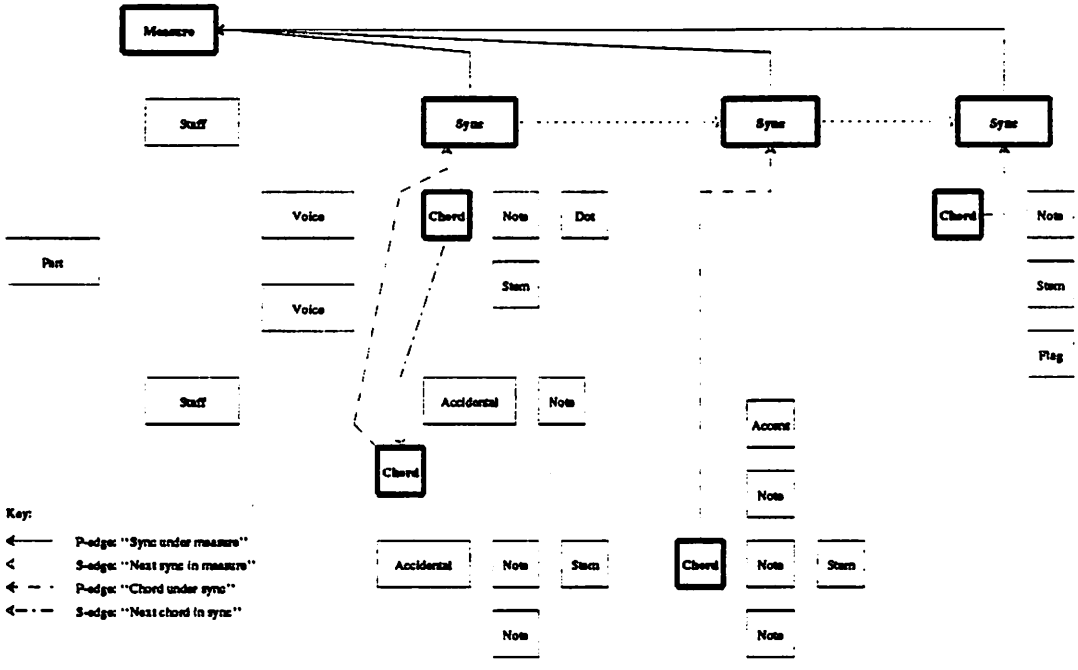


Figure 4.19. Orderings under the Measure and Its Syncs

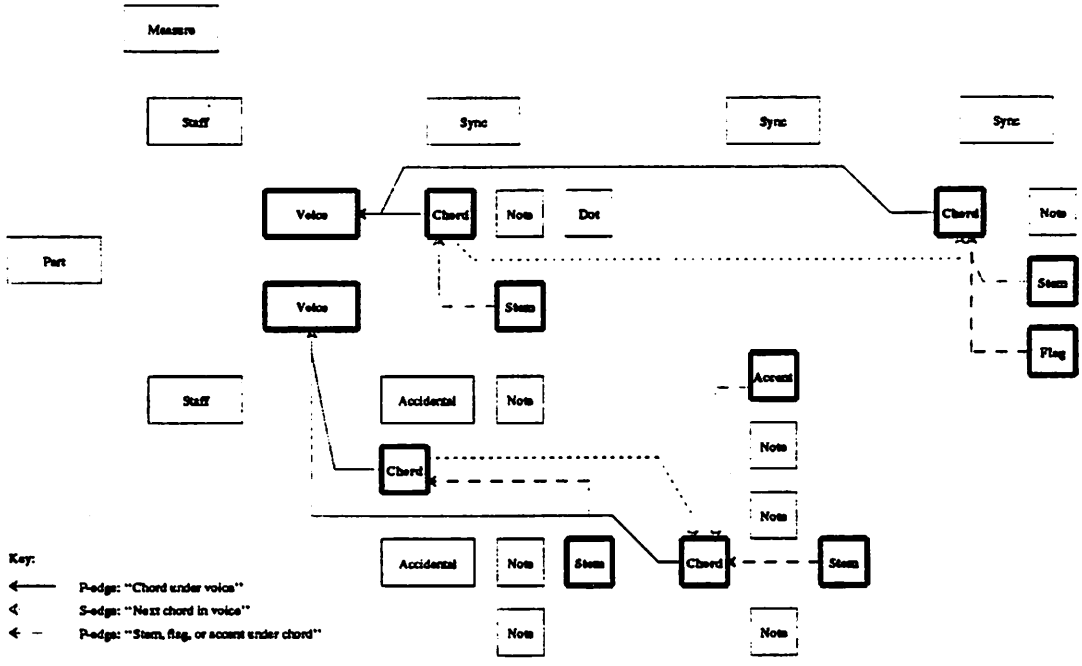


Figure 4.20. Ordering of Chords and Their Graphical Components

Similarly, figure 4.21 shows the ordering of notes within chords, and the orderings of accidentals and dots within notes. Again, accidentals and dots form ordered sets with a single element in this example, therefore no S-edges are shown for them. Finally, the ordering of notes under their parent staves is shown in figure 4.22.

Although this example is small, the instance graph is complicated. Tallying the objects manifest in this single measure (including some not included in the simplified instance graph above), there are a total of 60 database entity instances.

Even in this simple example, the complexity of the topological information inherent in CMN is evident. CMN has historically been read by human performers needing to derive large amounts of information from the score in "almost" real time. The emphasis for musical notation has been to provide the performer *at a glance* with virtually all the information necessary to render the music. The notation itself has therefore evolved over time to provide as concisely as possible a very large and diverse quantity of musical information, including pitch, rhythm, phrasing, dynamics (e.g. volume), harmonic structure, declamation (e.g. of syllables of text in a song), and so forth.

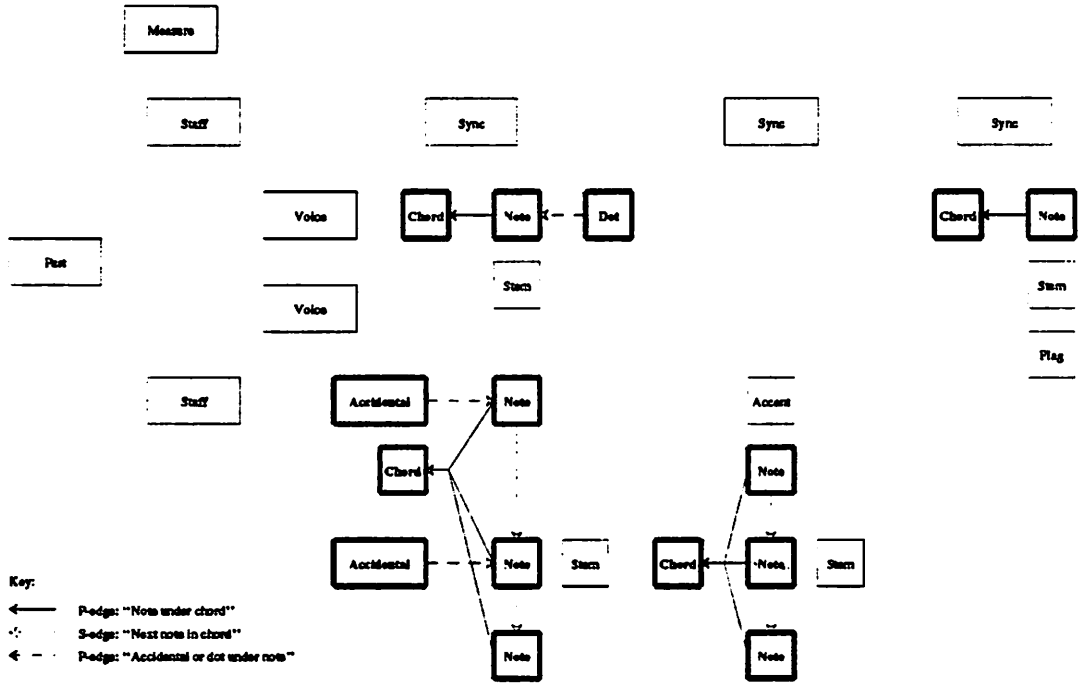


Figure 4.21. Ordering of Notes (by Chord) with Their Graphical Components

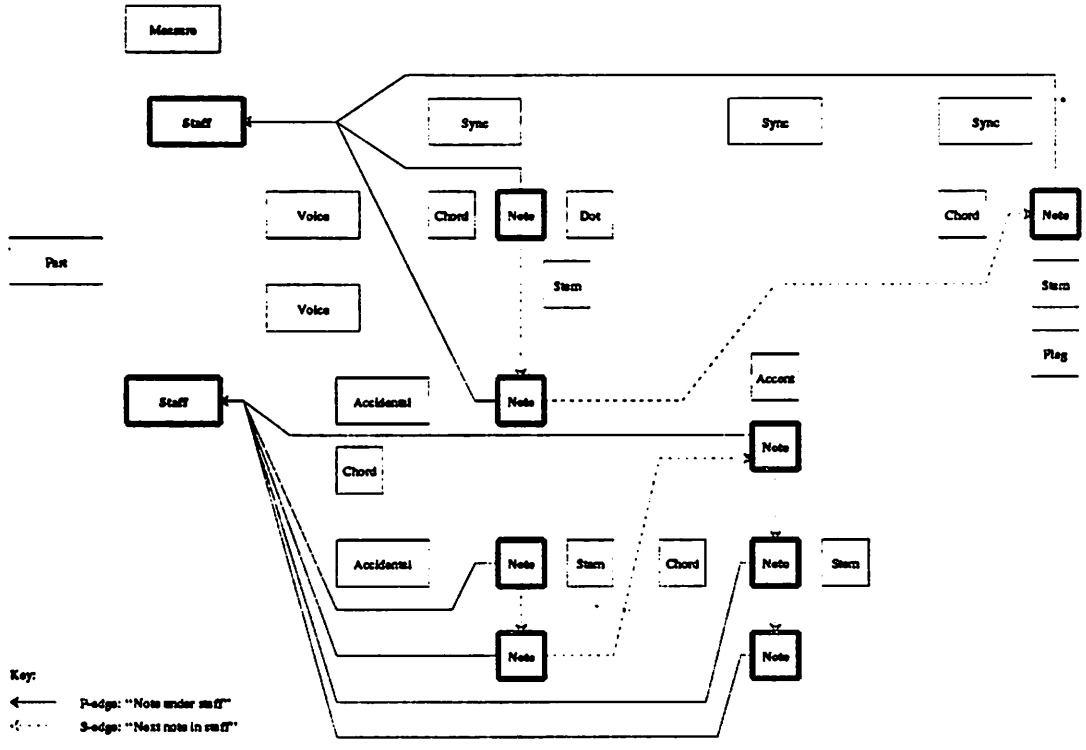


Figure 4.22. Ordering of Notes by Staff

4.5. Projecting the Size of Music Databases

Having determined the set of entities required to represent a CMN score, the size of this representation for a particular score instance may be predicted. This section considers the question, "How much information must a music data manager manage?" As with any general purpose data manager, the answer to this question is highly application dependent. In spite of this, some general characterizations about the size of music databases may be made that do not assume a particular application. An empirical analysis was performed in an indirect fashion, intending to derive an "order-of-magnitude" estimate of the size of musical objects. A simple model for predicting musical information density is developed later in this section, based on our results.

4.5.1. Counting the Entities of a CMN Score

The "unit of publication" (i.e. a book) was chosen as the musical object for analysis. The advantage of this is that the information density within a given book is fairly constant, while the infor-

mation density between different books representing, for example, different genres of music varies widely.

For the practical problem of determining the size of objects, a statistical approach was used. Assuming (very roughly, to be sure) that the density of musical information is uniformly distributed over the book, samples were taken on a page by page basis to extrapolate the total amount of information.

Our approach, then, was as follows: For each unit of publication, determine the number of each musical entity as defined by the schema of the previous section. This was done by taking a random sample of pages, counting the entities on those pages, and extrapolating the total number of entities from this count. In those cases where exact amounts were available (where, for example, measures or pages were already numbered), the exact values were used in preference to the statistical extrapolations.

Figure 4.23 shows our results for three different kinds of books. They are all similar in that they contain exclusively musical material. Other possibilities were not considered, such as thematic indices, which contain musical material embedded in a large amount of bibliographic (textual) infor-

Entities	Collection: Broadway Tunes	Opera: Die Fledermaus	Symphony: Beethoven's Fifth
performers	2	48	29
movements	69	18	4
pages	176	704	136
lines	900	704	186
measures	4,000	4,000	1566
parts	140	100	40
staves	200	300	30
voices	350	400	60
syncs	20,000	22,000	8,000
chords	52,000	520,000	96,000
notes	65,000	660,000	160,000
rests	5,000	58,000	80,000
beams	6,000	66,000	7,000
syllables	14,000	58,000	0
Total	170,000	1,400,000	350,000

Figure 4.23. Number of Entities in Musical Objects

mation.

The three books are:

(1) A collection of Broadway tunes, scored for voice and piano [Leo81],

(2) A three-act opera, *Die Fledermaus*, by J. Strauss [Str68],

(3) Symphony Nr. 5, by L. van Beethoven [Bee60].

As can be seen from the data in figure 4.23, the number of entities is dominated by the number of notes and chords.

4.5.2. Predicting Database Size

It would be useful to predict the database size for a given musical score based on a small set of readily determinable parameters, such as the length of the score, the number of parts, and so on. In this section, the relationship between these parameters and the resultant database size is determined.

The total number of entities, e, in a score will form the basis for our estimate of its size. This assumes that all entities are of about the same size. This is reasonable for the above examples, where at least two-thirds of the entities in a score are one of two types, notes or chords. By noting the duration (performance time) of a score, and the number of performers in the orchestra, the average density of the score can be calculated:

$$d_{avg} = \frac{e}{tp}$$

where:
d_{avg} is the score density in entities per performer per second
e is the number of entities
t is the duration of the score
p is the number of performers in the score

Given the density of a score (or perhaps, the density expected in a particular genre of scores), score size is determined by the formula:

$$s = tpd_{avg}s_{entity}$$

where:
s is the size of the score representation in the database
t is the duration of the composition in seconds
p is the number of performers in the score
d_{avg} is the average density (as above)
s_{entity} is the average size of an entity (assume 4 bytes)

The average density parameter and total size for our three sample scores are shown in figure 4.24, along with the rate of information flow during performance, measured in kilobytes per performance second.

We hypothesize that scores of a similar genre have similar densities. Indeed, the two classical orchestral piece considered above have densities of 3.6 and 4.0. The higher density, 10.6, of the piano vocal work can be explained by two factors. First, the piano is physically capable of producing multiple notes simultaneously (and, unlike, say, the violin, does so most of the time). Second, a composition for two performers will generally use both of them continuously, whereas in an orchestral composition, large subsets of the orchestra are often silent.

It would be interesting to test this hypothesis; to determine, once the music data manager is in operation, whether there is indeed a strong correlation between orchestration or compositional style and information density.

Gomberg [Gom77] did actually perform a DARMS encoding of a sophisticated orchestral composition. He encoded approximately twenty percent of Elliot Carter's "Double Concerto for Harpsichord and Piano with Two Chamber Orchestras" [Car62]. The approximate size of Gomberg's encoding is 500 kilobytes, with 3000 bytes per page, 300 bytes per performance second. According to the our model,

t = 1500 seconds (known)
p = 21 voices (known)

Entities	Collection: Broadway Tunes	Opera: Die Fledermaus	Symphony: Beethoven's Fifth
Entities e	170,000	1,400,000	350,000
Performance time t (seconds)	8000	8000	3000
Density d_{avg}	10.6	3.6	4.0
Size s (megabytes)	8	67	17
Kilobytes per performance-second	1	8	5

Figure 4.24. Projected Database Size

$d_{avg} = 4$ (assumed)
$s_{ent} = 4$ (assumed)
Therefore:
$s = 504$ Kilobytes

This result supports our hypothesis that density is consistent across scores of similar genre.

4.6. Summary

In the context of Common Musical Notation, the large number of entities that make up a conceptual representation of music have been explored. These entities have attributes that fall into one of three classes: temporal, timbral, or graphical. The timbral aspect of music entities can be subdivided into three more classes: pitch, articulation and dynamics. Within the graphical view, an important set of attributes relate to textual information.

The hierarchical ordering graph that represents the relationships among the entities within each of these aspects was presented. This entailed a careful enumeration and definition of all the entities that constitute our representation for CMN scores.

The information density of actual musical scores, based on a statistical analysis of the number of entities per unit of publication, was then calculated. This may be used to give a rough value of the expected size of an arbitrary composition. The most obvious conclusion from this analysis is that musical compositions have a very high information density. Any musical information management system that is to handle even single compositions of moderate complexity must manage a large quantity of information.

CHAPTER 5

An Access Method for Ordered Aggregation

An access method that is useful for manipulating ordered data will be described in this chapter. This access method, based on a data structure known as the A-tree, provides an efficient way to manipulate the ordering of entities within a relation. A design is presented for this access method as an extension to the INGRES database system, and to its data definition and manipulation language, QUEL.

The chapter begins by describing previous proposals for representing ordered relations in section 5.1, and user-defined aggregate functions in section 5.2. These two concepts serve as the starting point for our proposal.

This proposal supports inherited attributes whose values are determined by an aggregate function over the siblings that participate in an ordering. These *ordered aggregate functions* are described in section 5.3.

An ordered relation may be stored in either a flat file format (called an *ordered heap*) or in an A-tree structure. The specification of these is presented in section 5.4. In order to use this data structure, the database system makes use of procedures supplied by the user.[1] The procedures that the user must code in order to support an ordered aggregate function are described in section 5.5.

In section 5.6, we present the extensions to our data definition language needed to support A-trees. These commands allow the user to register ordered aggregates with the system, associate ordered aggregate functions with entities, and create A-trees over ordered relations.

The algorithms employed by the database system to manipulate ordered relations and determine the value of ordered aggregate functions are specified in section 5.7.

A prototype A-tree implementation has been built, and various performance tests have been performed. These are presented in section 5.8.

---

[1] It should be clear that the "user" in the context of "user-defined aggregates" is the application programmer developing a client for the musical data manager. Such a user should not be confused with the end-user of the application, for whom all database operations are presumably transparent.

Until this point, we have only been considering relations with a single, global ordering. Section 5.9 discusses the case where the entities in the relation participate in multiple orderings, and section 5.10 considers hierarchical orderings.

An extension to this implementation, supporting multiple orderings, is then explored. This involves storing a multi-ordered relation as an multi-linked list. This allows for a graph representation of the ordered data. This structure is often more space efficient, and graph partitioning techniques may result in superior query performance. These issues are detailed in section 5.11.

Section 5.12 briefly discusses other issues such as additional implementation alternatives and performance optimizations.

5.1. Previous Proposals for Representing Order

In this section, previous proposals for implementing notions of ordering will be discussed. They fall into two classes: sorted relations and ordered relations. The latter class contains several interesting proposals, such as ordered B-trees and event trees, which will be presented in detail.

5.1.1. Sorted Relations

Although the original description of the relational model specified that the records in a relation are not ordered [Cod70], implementations of relational databases have typically provided a means to sort the records of a relation by using one or more of its attributes as a "key." Relations may be stored in ISAM [IBM66] or B-tree [BaM72, Com79] data structures to allow efficient access via the key value, and to maintain the records in sorted order after insertions or deletions. Relations that are ordered by a key value are "sorted relations."

The ordering structures presented in chapter 3 are not well modeled by sorted relations. Because ordering in sorted relations is dependent on the value of a native key attribute, which doesn't exist for many ordered relations (as was shown in chapter 3), these techniques are not suitable for the music database. This shortcoming of sorted relations motivated the development of *ordered relations*.

5.1.2. Ordered Relations

A common implementation of the entity model, built on the relational model, puts each entity instance into a data record, and all the records of a single entity type in a single relation. If the entities are ordered, we can reflect that ordering by the order in which the records are situated in the relation. A relation whose records are ordered in this way is an *ordered relation.* The database system must properly maintain this ordering. For example, the system may not arbitrarily rearrange the records of the relation, as it might otherwise do if the relation were unordered.

The TEXT relation [SSK82] serves as a simple example of such an ordered relation. It models the lines of text in a document. Each record in the TEXT relation represents one line of text. These lines are obviously ordered; whether one line in a document is before or after another line is well defined.

Suppose we implement the TEXT relation as a sorted relation, by introducing the attribute "line number" as a key.[2] If the relation is then sorted on line number, the lines become properly ordered. This does not, however, accurately model the TEXT relation. As lines are inserted or deleted from this relation, the "line numbers" must all be changed, even when the order of the remaining objects remains the same. For this case where the records in a relation are ordered, yet no key properly determines the ordering, sorted relations are not appropriate. An ordered relation is used instead.

QUEL normally allows the comparison of attribute values in a query. In an ordered relation, the records themselves (as entities) may be compared. For example, a valid query on the ordered TEXT would be,

```
range of t1, t2 is TEXT
retrieve (t1.all)
  where t1 before t2
  and t2.line = "a line of text"
```

This query retrieves all the lines prior to the line, "a line of text." The range variables, t1 and t2 in this example, represent entities that may be compared. The comparison is well defined because the entities are ordered.

---

[2] In this model, a line number represents the ordinate position of a line within a document (for example, the fifth line in the document has line number five). Such line numbers are necessarily consecutive. This notion differs from the concept of line number used by some text manipulation systems.

5.1.3. Ordered B-Trees

A structure for implementing text as an ordered relation is presented in [SSK82]. In this proposal, each line of text is stored as a record in the TEXT relation. Because this proposal does not use an entity model, ordering had to be reflected in an attribute of the TEXT relation. The line number, associated with each line of text, is presented as such an attribute. Rather than having the user update every line number whenever an insertion or deletion is performed, an auxiliary data structure is used to maintain correct line numbers for each record without user intervention. This data structure is called an Ordered B-tree (OB-tree). it is presented briefly here; a complete description may be found in [Lyn82].

An OB-tree is similar to a B^+-tree [Com79] in that data is stored in leaf pages, and a multi-level index is provided to access the data. An example of an OB-tree is shown in figure 5.1. Pointers to the records in the relation (i.e. tuple identifiers, or *TID*'s) are stored in the leaf pages of the OB-tree. The order of the records is represented by the order in which the TID's are stored. Each internal record of the OB-tree maintains a count of the TID's in the subtree below it.

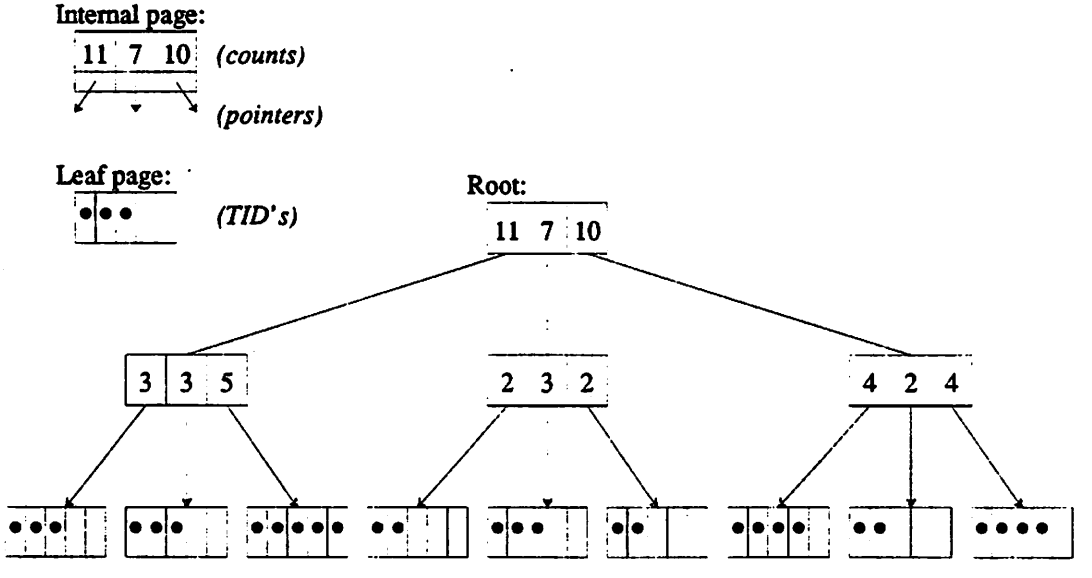


Figure 5.1. An OB-tree (from [StR80, p. 15])

In order to find a particular line, given its line number, the data manager scans the root of the OB-tree, determining which subtree contains the particular line. Suppose we wish to find line 15 using the OB-tree of figure 4.23. Scanning the root shows that the left subtree contains lines 1 through 11, and the middle subtree contains lines 12 though 18. We therefore must find the fourth line in the middle subtree, by scanning its root, and so on. One page is scanned at each level of the tree until a leaf page is encountered. The TID for the desired record is found on this page.

In order to insert or delete TID's from this data structure (corresponding to insertion or deletion of lines of text in the TEXT relation), algorithms similar to those for B^+-tree insertion and deletion are used. The important point is that for a single insertion, rather than updating all the line numbers which follow the insertion point, we merely need to update one value at each non-leaf level of the OB-tree.

5.1.4. Other Proposals for Ordered Relations

An almost identical approach is used in the EXODUS system [CDR86] for storing "large data objects." A large data object consists of a variable length string of bytes split among several disk pages. An OB-tree index provides efficient access to any substring at a given ordinate position, determined by a "byte number" instead of a "line number." Algorithms for inserting blocks of bytes at an arbitrary point in the string are presented in this proposal. In the OB-tree for a large storage object, the internal records of the index contain counts of the number of data bytes (rather than data records) in the leaf pages below.

In a proposal for managing events and processes, a similar extension to B-trees was presented [Rub85]. In this proposal, an event is an entity that begins at a particular point in time. They form an ordered entity set, such that an event e_1 is before another event e_2 in the ordering if e_1 starts at an earlier point in time than e_2.

Each event is stored in an EVENT relation, along with the amount of time until the start of its succeeding event (the *delay*). The start time of any event can be calculated by summing the delays from the beginning of the ordering up to the given event. An auxiliary tree index allows the start time to be calculated for a given record without requiring a sequential scan of every preceding record. For each record in the EVENT relation, its TID and delay are stored in this index, as shown in figure 5.2. Each internal record of the B-tree index contains the sum of all the delay values under it.

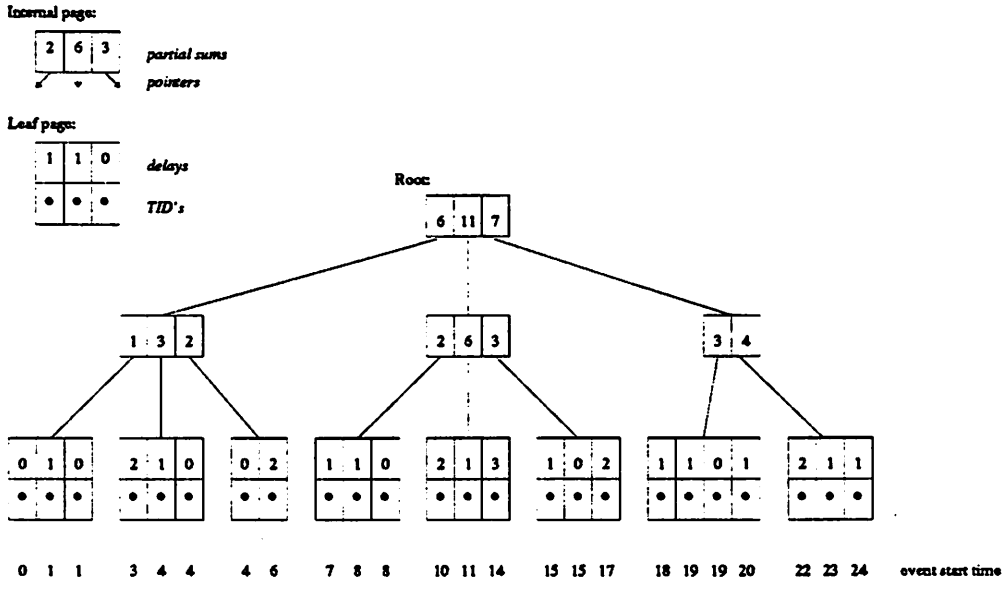


Figure 5.2. A Start Time Index for Events (from [Rub85, p. 15])

Referring to this figure, suppose we want to find the event at start time 20. We first scan the root. The first subtree under the root must contain start times within the range 0 through 6, the second subtree covers 6 through 17, and the third subtree covers 17 through 24. The event at start time 20 therefore must lie under the third subtree. Its left child is seen to contain start times in the range 17 through 20. We then scan this leaf page. Beginning with a start time of 17, we add the delays for successive records on this page, until we reach the desired start time of 20 at the fourth record. We then take the TID from this record, which points to the desired data record in the EVENT relation. Thus, rather than scanning every leaf page to determine the sum of the delay values, a traversal of the index from root to leaf suffices to find the event with a given start time.

Insertions and deletions are performed as for B-trees, with the addition that the internal partial sums must be properly updated.

5.2. User-Defined Aggregates

In the TEXT example, the index maintained a count of the data records in the ordering. In the EVENT example, it maintained a sum over the delays associated with each event. "Count" and

"sum" are each instances of aggregate functions provided by query languages such as QUEL. Our proposal generalizes these examples so that arbitrary aggregate functions may be incorporated into the index.

The INGRES system [HSW75] has been extended to provide a facility for defining abstract data types (*ADT*'s) and abstract data type operators [Fog82, Ong82, Ong83]. The result, ADT INGRES, has been further extended to incorporate user-defined aggregates, including aggregates over abstract data types [Han84]. Our proposal further develops this work by extending ADT INGRES to support user-defined ordered aggregates (to be defined presently) over ordered relations.

An example of an ADT is the data type *box* used for graphical descriptions (for instance, in a VLSI application, as described in [SRG83]). A straightforward implementation of the *box* type uses a group of four floating-point numbers to specify the vertices of a rectangle. An example of an ADT operator on boxes would be the *overlap* operator, "||". This operator takes two boxes as arguments, and returns **true** if they overlap. The following example (from [SRG83, p. 5ff]) demonstrates the use of this ADT operator:

```
create BOXES (owner = integer,
  layer = string,
  box_desc = box_ADT)

append to BOXES (owner = 99,
  layer = "polysilicon",
  box_desc = "0,0,2,3")

range of b is BOXES
retrieve (b.box_desc)
  where b.box_desc || "0,0,1,1"
```

First, the BOXES relation is created. It contains an integer attribute "owner," a character string attribute "layer," and an ADT attribute, "box_desc." The second statement inserts a record into the BOXES relation. The attribute value "0,0,2,3" is translated by the query parser into a box: a rectangle with one corner at x=0, y=0, and the opposite corner at x=2, y=3. The third statement retrieves all boxes that overlap the unit square (the square with one corner at x=0, y=0, and the other at x=1, y=1). The procedures to manipulate boxes and to implement the overlap operator are supplied by the user.

Hanson extends the notion of user-defined ADT operators to user-defined aggregates. He gives two examples of aggregates on boxes [Han84, p. 3ff]:

(1) Compute the area occupied by a set of possibly overlapping boxes. For example, find the area of boxes in the polysilicon layer:

```
range of b is BOXES
retrieve (area(b.box_desc where b.layer = "polysilicon"))
```

(2) Compute the smallest box containing a set of boxes (the bounding box). For example, find all boxes that overlap the bounding box of the polysilicon layer:

```
range of b1, b2 is BOXES
retrieve (b.all)
    where b.box_desc || bounding_box(b.box_desc where b.layer = "polysilicon")
```

Determining aggregate function values is performed by scanning sequentially through the records of the relation. At each record, particular values are passed to a routine which accumulates state information used to calculate the aggregate value. This routine is called the *Next* routine. For example, the aggregate *average(x)* (where x is an attribute in the relation being scanned) maintains two numbers in its state: the sum of x values, and a count of the number of records scanned. For each record scanned, the *Next* routine adds an x value to the sum, and increments the count. When the scan is complete, the average may be calculated by dividing the resultant sum by the count (this is done by another user-defined procedure, the *Result* routine).

By allowing the user to define the structure of this state information, as well as the routine used to incrementally accumulate this state, user-defined aggregate functions are supported. To process the bounding box aggregate function in a query such as:

```
range of b is BOXES
retrieve (bounding_box(b.box_desc where b.layer="polysilicon"))
```

the system sequentially scans through a set of boxes that satisfy the qualification (i.e. boxes in the polysilicon layer), and, for each box, calls *Next* to accumulate state information. The state, in this case, is initially an empty box (the "state box"). As each box is scanned by the system, the *Next* procedure extends the state box to cover this scanned box. When the scan is complete, the state box is returned to the data manager as the result of the aggregate calculation.

5.3. Ordered Aggregate Functions

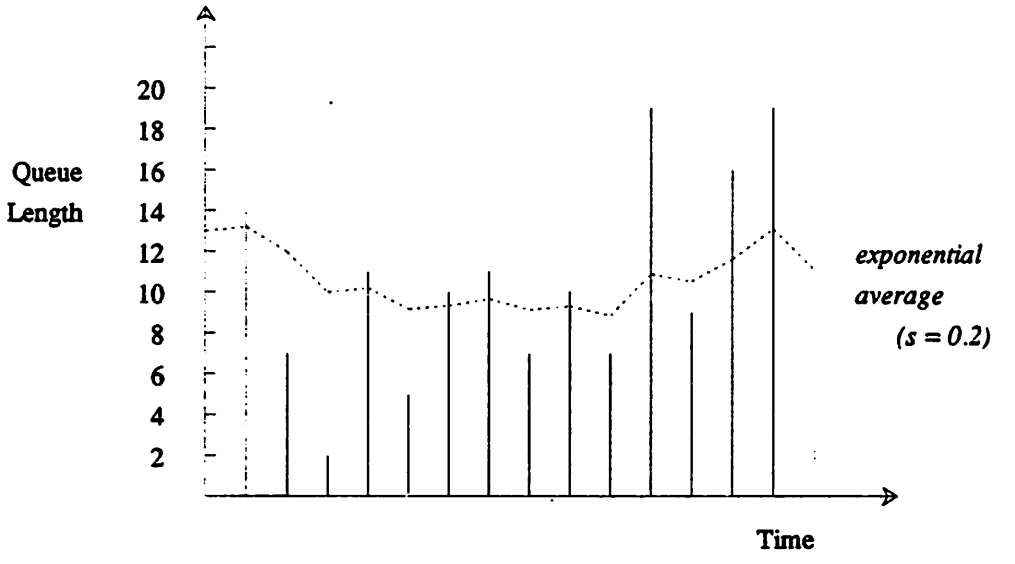
The aggregates described in the previous section take an unordered set of entities (e.g. boxes) and calculate a single value based on them (e.g. a bounding box). An *ordered aggregate function*, on the other hand, takes an ordered set of records, and defines one value for each record. In terms of the entities stored in an ordered relation, the ordered aggregate function defines an inherited attribute whose value for any particular entity instance depends on the position of that instance in the ordering. The following sections contain several examples of ordered aggregate functions.

5.3.1. Examples of Ordered Aggregate Functions

It can be seen that the line numbers of the TEXT relation are the result of such an ordered aggregate calculation. A line number for a given record is the "count" aggregate applied to the records previous to it in the ordering. This definition of line number remains correct in the face of arbitrary insertions and deletions from the TEXT relation. Similarly, the start time of an event in the ordered EVENT relation results from applying the ordered aggregate for summation to the *delay* values previous to this event in the ordering.

In addition to the ordered aggregates for counting and summation (hereafter called "ordered_count" and "ordered_sum," respectively), other application specific ordered aggregates might be useful. Suppose we wish to model a queue, whose length changes over time, as elements are added and removed from the queue. For example, a process scheduler may have a queue of runnable processes. At regular time intervals, we sample the length of the queue, and store it in the database. This set of queue length samples constitutes an ordered relation (each sample is an entity, and each entity is before or after some other entity). We want to know the average length of this queue in the vicinity of a given point in time. This is commonly known as the "load average" of the system.

The averaging function used for this example is typically an exponentially weighted average. The exponentially weighted average at a given point in time is a function of the queue length at all previous sample points. Samples in the recent past are more heavily weighted than samples in the distant past (The assumption in this example is that "recent history predicts future behavior"). Figure 5.3(a) shows a histogram of a queue length samples, and an associated exponential average. The exponential average serves as a "smoothing function" over the raw data samples. These samples are stored in the



(a)

RUNQUEUE		
Queue Length	Time (ordered count)	Load (exponential average)
13	0	13.00
14	1	13.20
7	2	11.96
2	3	9.97
11	4	10.17
5	5	9.14
10	6	9.31
11	7	9.65
7	8	9.12
10	9	9.30
7	10	8.84
19	11	10.87
9	12	10.50
16	13	11.60
19	14	13.08
3	15	11.06

(b)

Figure 5.3. Exponential Average of Queue Lengths

RUNQUEUE relation, shown in figure 5.3(b). The only native attribute in this relation is queue length. The time attribute is an instance of the ordered count aggregate, analogous to the "line number" attribute in the TEXT relation. The third column, the exponential average, contains the new ordered aggregate for exponential averaging. Its value at a given data point is calculated by taking a weighted average of the average at the previous data point and the data point itself (the weight, or scale factor, in this example is 0.2). The exponential average may be computed by the recurrence relation:

$$\bar{x}_0 = a_0$$
$$\bar{x}_i = sa_i + (1-s)x_{i-1}$$

where:
a_i is queue length at time i,
$\bar{x}_i$ is the exponential average at time i,
s is the scale factor for weighting the average ($0 < s < 1$).

It is a straightforward mathematical exercise to determine the algorithms for accumulating state information in a manner analogous to that used for the ordered_sum and ordered_count aggregates. This is included, for completeness, in appendix E.

5.3.2. Components of an Ordered Aggregate Function

The recurrence relation in the above example contains components common to every ordered aggregate function. These are:

An ordering. In the example, the ordering is that of the samples in the RUNQUEUE relation.

State information. This is x_i in the example. It need not be an atomic value, but may be an arbitrary structure. For example, we have seen that the regular *average* function has two elements in its state, a sum and a count.

Attribute parameters. These are the attribute values taken from the entities in the ordering, which are used to calculate the aggregate function value. In the example, the queue length, a, is an attribute parameter. In the ordered_sum aggregate, the attribute over which we are summing is the parameter. The ordered_count aggregate requires no such parameter.

Constants. In contrast to the parameters whose values vary within the computation of a particular ordered aggregate, there may also be constants associated with the aggregate calculation. For

example, over a given ordered set of queue length samples, there are many different exponential average functions, depending on our choice of the "scale factor" constant, s. At the time we define an inherited attribute to have the value of an ordered aggregate function, we fix the value of constants associated with that function. To be clear, it should be noted that these so-called "constants" may vary among different instances of the ordered aggregate, but within a single instance of an ordered aggregate function, the values of these constants are fixed.

A result. The result value of the ordered aggregate function at any point in the ordering is a function of the state at that point. In the exponential average, it is merely the state value, x_i.

5.4. Implementation of Ordered Relations

Having detailed the components of ordered entities and ordered aggregate functions, their implementation is now considered. The simplest implementation, the ordered heap, is discussed first, followed by a more complicated data structure, the A-tree, that provides better performance for calculating ordered aggregate functions, at the price of increased space and complexity.

5.4.1. Ordered Heaps

A simple implementation of the ordered TEXT relation stores the records of the relation on a doubly-linked list of disk pages, as shown in figure 5.4. This flat file structure is known as an *ordered heap*.[3] Within a page, the ordering of records is represented by their position. A record is before

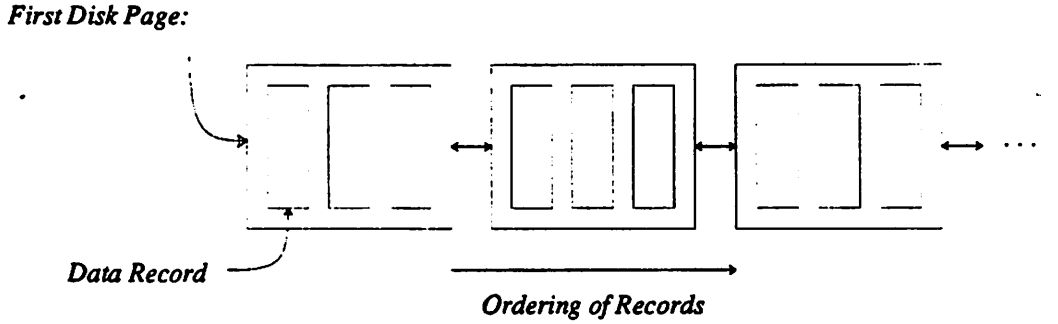


Figure 5.4. An Ordered Relation as a Linked List of Disk Pages

[3] The term "heap" is used to refer to an unstructured collection of objects (this usage is familiar in database systems, as well as in programming languages, such as Pascal). This should not be confused with the data structure, also known as a

another record on the same disk page if it is stored closer to the beginning of the disk page. In comparing records on different pages, the ordering is represented by the links. All the records on one page are before those records on the pages following it along the forward links of the list. A sequential scan of an ordered heap therefore retrieves its records in order.

5.4.2. The A-tree Data Structure

Just as sorted relations may be maintained simply in heaps or efficiently in B-trees, ordered relations may be maintained either in ordered heaps or ordered trees. The following proposal for ordered trees generalizes those mentioned above.

The data structure used to index ordered relations with ordered aggregates is a tree of disk pages, as shown in figure 5.5. The disk pages are divided into *internal* and *leaf* pages. Each record in an internal page contains state information for one or more ordered aggregates, plus a pointer to a child page. Each record in a leaf page contains one data record from the relation (or possibly a TID pointing

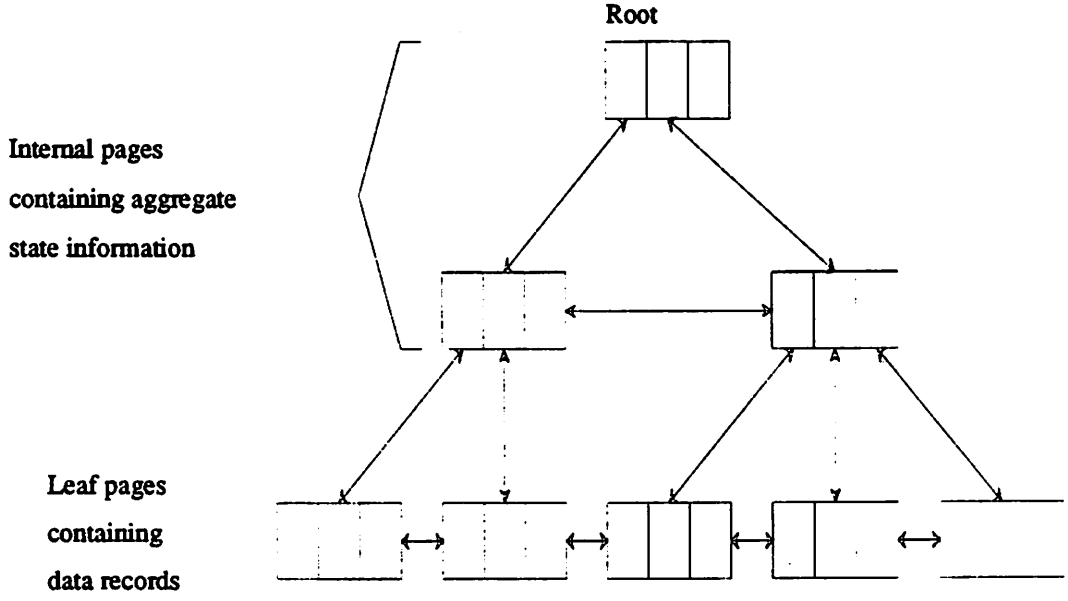


Figure 5.5. The A-tree Data Structure

heap, used by algorithms such as "heap sort", for which a heap is a partially ordered binary tree.

to the data record; this is discussed in section 4.9). Additionally, each page in the data structure (except for the root) has a pointer to a parent internal record. Finally, all pages within a level of the tree are doubly linked so they may be efficiently scanned, either forward or backward.

The data structure has the following characteristics:

All leaf nodes are at the same distance from the root of the tree. This is guaranteed by the insertion and deletion algorithms, as with B-trees.

All pages are at least half full, except possibly the root. In other words, if p is the number of bytes on a page, and r_i is the size of the i'th record (the records need not all be the same size), then the total size of records on a non-root page is bounded above and below:

$$\frac{p}{2} \leq \sum_i r_i \leq p$$

The ordering of entities in the ordered relation is reflected in the layout of records on the leaf pages. Scanning the leaf pages (following their forward pointers) accesses the data records in order.

The internal pages of the A-tree contain summary information for any number of ordered aggregates that have been associated (by **define inheritance** statements) with the ordered relation. The Ordered B-tree is thus a special case of this data structure, where the summary information in an internal record consists of the count of the leaf records rooted at that internal record. The index used for event start times is another special case of an A-tree, where the summary information for an internal record is the sum of a particular attribute (e.g. the *delay* attribute) over its subtree.

Note that the A-tree index provides no additional information beyond what is present in the ordered set of data records. Ordered aggregates such as ordered_count and ordered_sum may be determined by a sequential scan of the data. The index serves as a mechanism to improve on the amount of computation and on the number of page accesses required by the sequential scan. The internal pages effectively supply summary information that precomputes the aggregate over subsets of the data.

5.5. User-coded Routines

In order to implement a particular ordered aggregate, the user must supply the database system with a set of routines that determine how the aggregate is to be computed. Every calculation of an ordered aggregate value for a given entity (i.e. every query that references an attribute calculated using an ordered aggregate) must "scan" the ordered relation up to the given entity in order to perform the calculation. In the absence of an index, this scan is performed sequentially over the ordered heap. Five routines are needed to implement this scan. We now describe each routine, and, as an example, show their implementation for the ordered_sum ordered aggregate.

InitializeScan: The InitializeScan routine is called at the start of each scan. It allows the user to allocate the space for aggregate state information, and initialize that state.

For the ordered aggregate "sum," the state consists of a running sum over the course of the scan. It therefore consists of a single integer, and is initialized to zero.

NextInner: The summary information stored in an inner (non-leaf) record is simply the cumulative state for the subtree rooted at that record. Thus, as the scan passes through an inner record, NextInner is called with three parameters: the cumulative state (up to this record), the state for this record, and any constants associated with the ordered aggregate calculation (the scale factor seen in the exponential average ordered aggregate is an example of such a constant). The result of this routine is the cumulative state up to and including the inner record.

For the ordered_sum aggregate, the summary information consists of the sum over some data field in the leaf records rooted in a particular internal record. Thus, given a running sum (the current state), and summary information (from the internal record), we return a new current state whose value is these two states added together.

NextLeaf: The manner in which leaf records are accumulated into the state is generally different than the way in which internal records are accumulated. NextLeaf performs this function, given three parameters: the cumulative state and constants, as before, and a list of attribute values from the leaf record that enter into the aggregate calculation. The function accumulates these values into the state, and returns the updated state.

The ordered_sum aggregate has a single attribute parameter, the attribute over which we are summing (this was the *delay* attribute in the EVENT example). The new state for a given leaf record is the old state plus the value of the attribute parameter in this record.

Result: After the scan is completed, the state must be converted into a result value using this routine. For the ordered_sum aggregate, the accumulated state is the sum itself, so this is simply returned as the resultant value.

Compare: Under certain circumstances the user must provide a routine which takes two resultant aggregate values and determines their relationship one to the other. The *Compare* routine takes two results and returns a negative number, zero, or a positive number, depending on whether the first result is less than, equal to, or greater than the second result (respectively).

5.6. Defining Ordered Aggregates and A-trees

We now discuss further extensions to our DDL to provide the user with the commands to define aggregate functions and build A-trees over ordered relations.

5.6.1. Registering Ordered Aggregates with the Data Manager

A set of these routines for a given ordered aggregate function may be bundled into an executable file, to be loaded on demand by the database system (the mechanism for accomplishing this in ADT INGRES is discussed in [Fog82]). The user must inform the system of the existence of this file. The command to register an ordered aggregate function (similar to that used to register user-defined aggregates in [Han84]) is:

```
ordered_aggregate_definition:
    define ordered aggregate aggregate_name
        [ ( parameter { , parameter } ) ]
        returns type
        [ ascending | descending ]
        file = file_name

parameter:
    formal_parameter_name = [ constant ] type
```

This definition associates the name of an ordered aggregate ("aggregate_name") with the file that contains the executable routines implementing that aggregate ("file_name"). The parameters in this specification are formal parameters, much like those that would be used at the head of the routines

that implement the ordered aggregate. When records are scanned to evaluate an ordered aggregate, parameters which are to be fixed during the scan are declared to be **constant**. The remaining parameters will be assigned values from the attributes of the data records (as described in the next section).

The type produced by the *Result* routine is specified in the **result** clause. Both parameter and return types may be arbitrary ADT's. For example, the ordered sum aggregate sums over integers, and returns an integer result. It is specified by the statement:

```
define ordered aggregate ordered_sum
    (summand = integer)
    returns string
    file = "/aggregates/osum.o"
```

As an aside, Hanson's proposal also suggests the generalized type **numeric** to indicate an arbitrary numeric type, and the return type **typeof**(formal_parameter_name) to indicate a return value whose type matches a particular parameter type. These both would be appropriate, for example, in the ordered_sum aggregate, since one may sum over integers or floating point numbers, and the type of the result matches the type of the summand.

Certain ordered aggregates generate values that are guaranteed to be monotonically increasing or decreasing over ordered records. The **ascending** and **descending** keywords indicate this to the data manager, to allow for more efficient processing (as will be demonstrated when the actual traversal algorithms are discussed later in this chapter). The ordered_count aggregate has this property:

```
define ordered aggregate ordered_count
    returns integer
    ascending
    file = "/aggregates/ocount.o"
```

The ordered_count aggregate takes no parameters, and produces an integer result that is guaranteed to ascend monotonically over the ordering. Notice that the ordered_sum aggregate cannot use the **ascending** clause, as both positive and negative integers may be added to the sum. The running summation may therefore increase or decrease.

5.6.2. Associating Ordered Aggregates with an Ordered Relation

An ordered aggregate attribute is a special form of inherited attribute. In chapter 3, we formulated a general syntax for representing inherited attributes. Using that syntax, we associate line

numbers with the TEXT relation as follows:

```
define entity TEXT (line = string)

/* declare the elements of the TEXT relation to be ordered */
define ordering (TEXT)

range of t is TEXT
define inheritance t (line_number = ordered_count(t))
```

This last statement is equivalent to:

```
range of t,t1 is TEXT
define inheritance t (line_number = count(t1 by t where t1 before t))
```

The shorter syntax serves two purposes: it is more concise and readable, and the query processor can easily detect ordered aggregates on which it can apply the processing optimizations developed in this chapter.

5.6.3. Creating an A-tree Index

After the inherited attributes for an ordered relation have been defined, the user may specify that an A-tree is to maintained over the relation. This is done using the **modify** command:

```
modify relation_name to A-tree
```

The relation indicated by "relation_name" must be an ordered relation. If the relation already contains records, then the internal pages of the A-tree are created using this algorithm:

Create A-tree Index

R is the relation to be indexed.

1. If *R* contains a single page, then make that page the root of the A-tree, and stop
2. Create a new, empty internal page and make it the root
3. $p \leftarrow$ the first disk page of *R*
4. Create a new, empty internal record, r_p
5. Set the child of r_p to be p
6. Set the parent of p to be r_p
7. Insert r_p into the root page
8. $p' \leftarrow$ the page after p
9. Create a new, empty internal record, $r_{p'}$
10. Set the child of $r_{p'}$ to be p'
11. Set the parent of p' to be $r_{p'}$
12. Insert $r_{p'}$ after r_p

Insertion (steps 7 and 12) involves updating the summary information of the internal records in the A-

tree. Also, if the root becomes full, it is split, increasing the height of the tree. These are both performed as part of the insertion algorithm, presented in the next section.

5.6.4. Defining Order Using Sort Keys

Because a sort key defines an order on a set of relations, we provide a function to globally order a relation using a sort key. This is accomplished with the **reorder** command. A set of ordered entities containing the sort key must be defined, for example:

define entity TEXT (initial_line_number = **integer**)
define ordering (TEXT)

reorder TEXT **on** initial_line_number

In this example, the **reorder** command will sort the set of TEXT records on their initial line number, and define the ordering of TEXT records to be that induced by the sort. Subsequent insertions could then be performed (using the **before** and **after** clauses) against the newly generated ordering.

In general, this approach appears to provide a convenient means of establishing an ordering among a large number of existing records.

5.7. Retrieval from Ordered Relations

Queries that involve ordered relations must support an extended set of retrieval operations beyond those of ordinary relations. For a particular query, the following cases may occur.

- The **before** and **after** operators may appear in the qualification of the query.
- Ordered aggregate attributes may appear in the target list and/or the qualification of the query.

We now consider the implementation to support each of these cases.

5.7.1. Implementing the Before and After Operators

In the absence of an index, the **before** and **after** operators may be implemented using a sequential scan over an ordered heap. Given two records r_1 and r_2, the records may be compared as follows:

Comparison using Sequential Scan of an Ordered Relation

1. If r_1 and r_2 are not in the same relation, return "not comparable"
3. $p_1 \leftarrow$ the page on which r_1 lies
2. $p_2 \leftarrow$ the page on which r_2 lies
4. If $p_1 = p_2$, then
5. If $r_1 < r_2$, return "before"
6. If $r_1 = r_2$, return "is"
7. If $r_1 > r_2$, return "after"
8. $p \leftarrow p_1$
9. If p is the last page, return "after"
10. $p \leftarrow$ the page after p
11. If r_2 is on p, return "before"
12. Go to step 10

This routine returns one of {not comparable, before, is, after}. A qualification such as "x **before** y" is met if and only if the comparison algorithm returns "before." The comparisons in steps 5-7 compare the locations of the records on the page. If the r_2 is not on the same page as r_1, the algorithm searches subsequent pages (following the forward links in step 10) for r_2.

This algorithm for evaluating the order operators is unfortunately rather inefficient, in that it may require a sequential scan of the ordered relation (steps 10-12) to evaluate the order operators. In the presence of an A-tree, the order operators may be evaluated much more efficiently. The algorithm for comparing two records using an A-tree, is:

Comparison using A-tree Traversal

1. If r_1 and r_2 are not in the same relation, return "not comparable"
2. $p_1 \leftarrow$ the page on which r_1 lies
3. $p_2 \leftarrow$ the page on which r_2 lies
4. If $p_1 = p_2$, then
5. If $r_1 < r_2$, return "before"
6. If $r_1 = r_2$, return "is"
7. If $r_1 > r_2$, return "after"
8. $r_1 \leftarrow$ parent of p_1
9. $r_2 \leftarrow$ parent of p_2
10. Go to step 2

Again, the comparison operators in steps 5-7 compare the locations of the records with respect to the beginning of the page. The loop in steps 2-10 is performed, at most, once per level of the A-tree, since p_1 will equal p_2 when they reach the root, at which point the algorithm terminates.

5.7.2. Implementing Retrieval of Ordered Aggregate Attributes

When an ordered aggregate attribute appears in the qualification of a query, the records of an ordered relation must be scanned to determine the value of the ordered aggregate associated with a particular record, and if that value satisfies the qualification.

There are three types of scans which the system must implement in order to support this operations: sequential scan, top down A-tree traversal and bottom up A-tree traversal. For reference, we will use this query as an example:

```
retrieve (TEXT.line)
    where TEXT.line_number < 10
```

In this query, the ordered aggregate attribute appears in the qualification, compared to a constant value. Queries can in general be reduced to this form by the process of *query decomposition* [WoY76]. Such simple queries are known as "one variable queries." We will call the constant in the qualification the "search value."

5.7.3. Sequential Scan

When the system needs to determine the value to be associated with a given ordered aggregate attribute in a one variable query, the following algorithm is used:

s is the current state,
p is the current disk page,
r is the current record.

1. $s \leftarrow InitializeScan()$
2. $p \leftarrow$ the first page of the ordered relation
3. For each record r in p:
4. $s \leftarrow NextLeaf(s,r)$
5. Substitute $Result(s)$ for the aggregate attribute in the query,
   and the values in r for the remaining attributes
6. If the query qualification is satisfied, return the values
   in the target list

For certain qualifications, the loop at step 3 may be terminated prematurely if the ordered aggregate function was registered using the **ascending** or **descending** clauses. For example, we have seen that the ordered_count aggregate is defined as **ascending**. Thus, in our example query,

```
retrieve (TEXT.line)
    where TEXT.line_number < 10
```

the algorithm could stop scanning the TEXT relation after it found a line_number equal to 10, because the system is guaranteed that all future line numbers are greater than 10, and thus do not satisfy the query.

5.7.4. Top-Down Traversal

Top-down traversal is used on ordered relations that have an A-tree index, to efficiently determine the record that has a given ordered aggregate value. It is only useful for aggregates that are **ascending** or **descending**. Our example query has these properties, and is resolved by the following steps:

Top-down Traversal

x is the search value,
s is the current state,
p is the current disk page,
r is the current record.

1. $s \leftarrow$ *InitializeScan*()
3. $p \leftarrow$ the root page of the A-tree
4. If p is an internal page:
5. For each internal record r in p:
6. $s \leftarrow$ *NextInner*(s,r)
7. if $x <$ *Result*(s) then
8. $p \leftarrow$ child of r
9. Go to step 4
10. Otherwise p is a leaf:
11. For each leaf record r in p:
12. $s \leftarrow$ *NextLeaf*(s,r)
13. if $x \geq$ *Result*(s) then
14. Mark record r and stop

At the end of this procedure, some leaf record has been marked. If the qualification is of the form "attribute = x," or "attribute > x", then all the records satisfying this qualification are at or beyond the marked record, and a sequential scan may be performed starting at the marked record. If the qualification is of the form "attribute < x" (as in our example query), a sequential scan from the beginning of the relation up to the marked record retrieves all the qualifying records.

If the ordered aggregate is defined to be **descending**, then the sense of the comparisons in steps 7 and 13 of the above algorithm must be reversed.

5.7.5. Bottom-Up Traversal

Under certain circumstances, the query processor, given a record in an ordered relation, needs to find the value of an ordered aggregate attribute for that record. This would occur in the processing of a query against the RUNQUEUE relation (described in section 5.2.1, above) to find the load at a point in time:

```
range of q is RUNQUEUE
retrieve (q.load)
    where q.time = 5
```

This is processed by performing a top-down traversal of the A-tree to find the record that satisfies the qualification, namely the record whose ordered_count attribute "time" has a value of 5. Given this record, the query processor needs to find its value for the ordered aggregate attribute "load." This is done using a bottom-up traversal of the A-tree. Notice, in this example, that the same A-tree is used for both traversals. Each internal record of the A-tree contains the summary information for both the "time" attribute (an ordered_count value) and the "load" attribute (an exponential_aggregate value).

The following algorithm is used to perform a bottom-up traversal from record r':

Bottom-up Traversal

s is the current state,
p is the current disk page,
r is the current record.

1. $s \leftarrow$ *InitializeScan*()
2. $r \leftarrow r'$
3. $p \leftarrow$ the page on which r lies
4. For each record in p up to (but not including) r:
5. If p is an internal page, $s \leftarrow$ *NextInner*(s,r)
6. If p is a leaf page, $s \leftarrow$ *NextLeaf*(s,r)
7. If p is the root, return *Result*(s)
8. Otherwise, $r \leftarrow$ parent record of p
9. Go to step 3

This algorithm returns (in step 7) the value of the ordered aggregate for record r'.

5.7.6. Updating Ordered Relations

For insertion into ordered relations, the syntax of the append command is extended as follows:

append to *relation_name*
[**after** *range_variable* | **before** *range_variable*]
(*target_list*)
where *qualification*

The standard form, **append to** *table_name* now has two additional clauses, to append before or after records in an ordered relation. As an example, to insert a line of text prior to the eighth line of text, one would say:

```
range of t is TEXT

append to TEXT before t (line = "inserted line of text")
    where t.line_number = 8
```

Although the qualification in the above example selects a single record, this need not be the case. For example, we might want to insert a line of text prior to every line that has the word "special" in it:

```
append to TEXT before t (line = "mark following line")
    where t.line = "* special *"
```

The asterisks in the string "* special *" cause the qualification to select every line that contains the word "special" anywhere within it.

In the above example, a single record is inserted repeatedly into a relation. It is also possible that several different records may be inserted into an ordered relation at a single point. For example, the following query inserts the first ten records of the NEWTEXT relation after line 5 of the TEXT relation.

```
range of t is TEXT
range of new is NEWTEXT

append to TEXT after t (line = new.line)
    where t.line_number = 5
    and new.line_number < 10
```

Because NEWTEXT is an ordered relation, the data manager must perform the insertions so as to assure that the order of the inserted records is preserved. In order to do this, the complete set of insertions must be determined. In the above example, this involves retrieving the first ten lines of NEWTEXT into a temporary ordered relation. Then, the data manager performs a bulk insertion of the ordered temporary relation into the TEXT relation. The algorithms for bulk insertion are given in

[CDR86].

Replace commands also may take an **after** or **before** clause, which move existing records within the ordering. For example, the following query moves lines 2 through 6 after line 7.

```
range of t1, t2 is TEXT
replace t1 after t2
    where t1.line_number >= 2
    and t1.line_number <= 6
    and t2.line_number = 7
```

Notice that the target list for such a replace may be missing, as in this example.

An update of this form must force the actual record to be moved (for this reason, it is similar to an update of a key field in a sorted relation). It is processed by retrieving the old values into a temporary relation (as for **append**), deleting them from the original relation, and re-inserting them at their new location.

A form of bottom-up traversal is used when updates are performed on an ordered relation that contains an A-tree. Any time a record is inserted, deleted, or modified on a leaf page, the summary information in its parent page (and their parent pages) must be updated.

To correct this summary information when a page is modified, we rescan the entire page with *NextLeaf* or *NextInner*. This results in a cumulative state that is then inserted into the parent record for that page. The parent page is thus modified, and the correction percolates recursively to the top of the tree. The algorithm for correcting the summary information after a modification to page p' is as follows:

Updating Summary Information

s is the current state,
p is the current disk page,
r is the current record.

1. $s \leftarrow$ *InitializeScan*()
2. $p \leftarrow p'$
3. If p is the root, then stop
4. For each record r in p:
5. If p is an internal page, $s \leftarrow$ *NextInner*(s,r)
6. If p is a leaf page, $s \leftarrow$ *NextLeaf*(s,r)
7. $r \leftarrow$ the parent record of p
8. Set the "summary information" of r to s
9. $p \leftarrow$ page on which r lies"
10. Go to step 3

These top-down and bottom-up traversal routines suffice to maintain the A-tree under retrieval and update. Two additional routines are necessary for insertion and deletion, namely those for splitting and merging pages.

5.7.7. Splitting Pages

It may happen, after an insertion of a record onto a page, that the set of records now on the page is larger than the page size. In such an instance, the page must be split. This is done as follows:

Split a Page

p is an overfull page.

1. If p is the root, split the root (see below), then go on to step 2
2. Determine r_p, the record in the parent of p that points to p
3. Create a new, empty page, p'
4. Move the latter half of the records from p into p'
5. Create a new, empty internal record $r_{p'}$
6. Set the child pointer of $r_{p'}$ to p'
7. Set the parent pointer of p' to $r_{p'}$
8. Insert $r_{p'}$ after r_p (recursively)
9. Update the summary information for pages p and p'

Because this algorithm is invoked in response to an insertion (which caused a page to become overfull), the subsequent insertion into the parent page is recursive. It may, in turn, cause the parent page to become overfull. The parent page would then be split, and so on.

In the case that the root is overfull, it must be split. This is accomplished by the following algorithm:

Split the Root

p is the overfull root.

1. Create a new page, which will be the new root, p'
2. Create a new, empty internal record, r_p
3. Set the child pointer of r_p to p
4. Set the parent pointer of p to r_p
5. Insert r_p into the (empty) root page p'

At this point, p can be split using the previous algorithm, since it is no longer a root node.

5.7.8. Merging Pages

After a deletion, a page may be less than half full of records. This violates a constraint on the A-tree structure, and the following algorithms correct this situation. The merge operation for deletion is analogous to the splitting operation for insertion. It operates as follows:

Merge Page

p is a page that is less than half full.

1. If p is a non-leaf root page, and p contains a single record,
2. Delete the root (see below), then stop
3. If there is no page after p
   $p \leftarrow$ *the page before* p (which necessarily exists)
4. Set p' to be the page after p (which now exists)
5. Let r_p be the parent record that points to p
6. Let $r_{p'}$ be the parent record that points to p'
7. If the records of p and p' can all fit on one page,
8. Append all the records of p' to p
9. Free the page p'
10. Delete (recursively) $r_{p'}$ from the parent of p, then stop
11. Otherwise,
12. Delete records from the beginning of p', and append them to p, until the two pages are equally full
13. Update the summary information above p, and above p' if it exists

The effect of this procedure is to make sure, after deletion, that all pages are at least half full. After steps 8-10, the single merged page must be at least half full because p' was at least half full. After step 12, both pages p and p' must be at least half full because the records in the two pages together take up more than one page (guaranteed at step 7). Thus dividing the records evenly results into two sets ensures each set requires no less than half a page.

Analogous to the operation of splitting the root on insertion, is the case where a root can be deleted. This happens when the root is not a leaf, and after deletion contains only a single internal record. The algorithm is:

Delete Root

Given: A root page p that is not a leaf,
containing a single record.

1. Set c to be the sole child of p
2. Free page p, and set the new root to be c

5.7.9. Operators over Ordered Entities: first and last

The system provides two additional aggregate functions that evaluate over ordered data. The first aggregate selects the first element in an ordering, and last selects the last one. These are provided by the system, rather than being implemented as ordered aggregate functions by the user, because:

- They have restricted semantics that typically allow for efficient implementation within the system as a special case (bypassing the scan and traversal algorithms).
- They result in an entity, rather than an attribute value, and are thus distinguished syntactically from other aggregate functions.

The system catalogs should include pointers to the first and last records of each ordering in order to efficiently support these aggregates. Certain systems actions, such as inserting records at the end of an ordering when no location is specified by the client, would make use of these pointers.

For example, to retrieve the line number of the last line in the TEXT relation, the command is:

retrieve (last(TEXT).line_number)

As with other aggregates, the value of the aggregate is calculated independently from the query as a whole. The aggregate value is then substituted into the query as a constant. In this case, the constant is an entity (syntactically, a range variable).

In the above example, because the system catalogs maintain a pointer to the last record in an ordered relation, the value of the line_number is readily accessible by doing a bottom-up traversal of the A-tree from this last record (assuming that the A-tree exists).

As with ordinary aggregates, It is possible that these aggregates may be qualified. For example, we may retrieve the first record containing a particular line:

retrieve (first(TEXT where TEXT.line = "marker").line_number)

In this case, the TEXT relation must be scanned for qualifying records (those that contain "marker"), and the first one returned. For the last aggregate, the relation could be scanned backwards.

5.8. A-tree Performance

We expected the performance of A-trees, in terms of storage utilization and overhead, to be equivalent to that of B-trees. A simulation of the A-tree algorithms was coded in LISP, in order to better understand the operation of the algorithms and determine their performance. This initial hypothesis was born out. The experiment performed 10,000 A-tree operations: first inserting 5000 random records, then deleting those records in random order. At each step, page utilization and page fault behavior were monitored. The simulation used a page size of 1024 bytes, leaf and inner record sizes of 16 bytes, and a buffer pool of 20 pages. Our results are summarized in figure 5.6, showing the number of pages, tree height, utilization, and number of faults per operation over the course of the experiment.

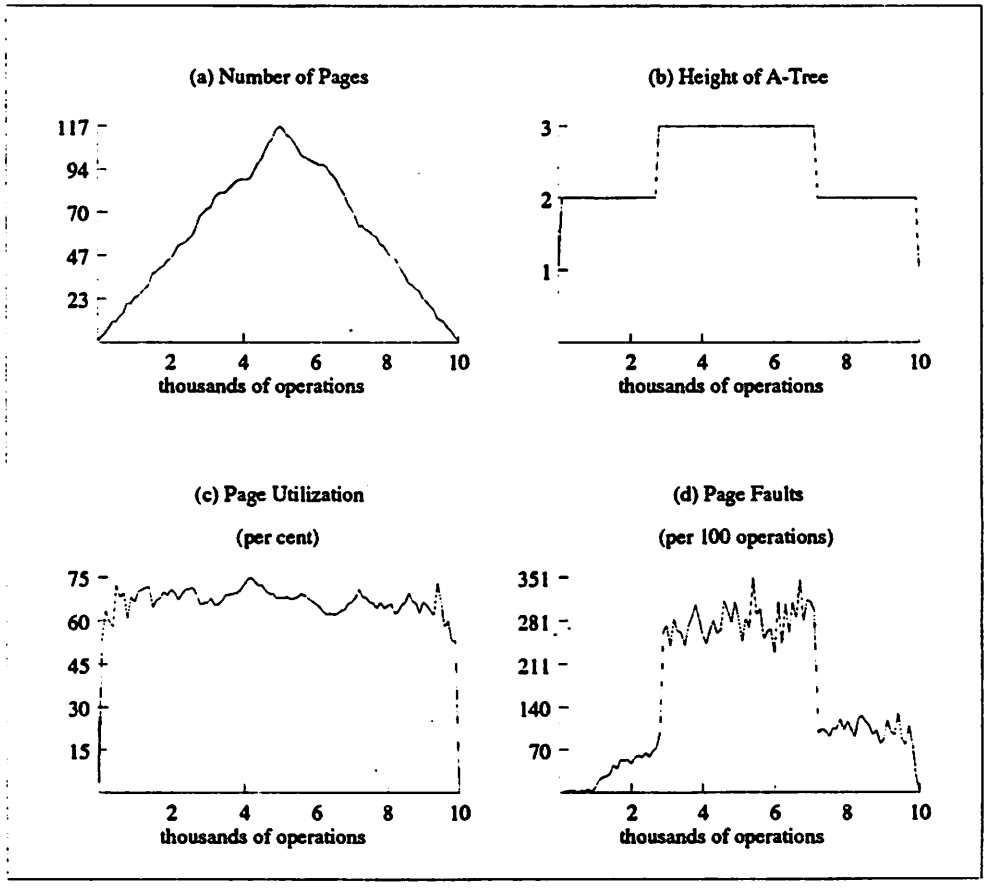


Figure 5.6. A-tree Performance (5000 insertions followed by 5000 deletions)

The first graph shows the size of the data structure, in pages, after each operation. At 16 bytes per record, a 1024 byte page contains at most 64 records. The number of pages increases linearly during insertion, and decreases linearly during deletion. The data structure requires 117 pages to store 5000 data records and 116 internal records.

The second graph shows the height of the A-tree over the course of the experiment. The root is split after insertion operations 65 and 2786. The height of the tree then remains constant until a total of 5000 records are inserted. During the subsequent deletions, the height of the tree decreases when the record drops to 2896 (at operation 7104), and again when it drops to 64 (at operation 9936).

Page utilization, shown in the next graph, is defined to be the number of bytes used for leaf records, divided by the number of bytes in the data structure. In our experiment, it turns out that less than three per cent of the pages are internal, so that almost all the unutilized page space is in the leaves. Except when the tree is nearly empty, the page utilization of the A-tree remains fairly constant, in the vicinity of 69%. This agrees with the analytical result for B-trees derived by Yao [Yao78].

The final graph indicates the number of page faults per 100 operations. A small buffer (20 pages) was used in order to improve performance for access that was restricted to a small number of pages. For the first 800 insertions, the entire data structure fit in the buffer, and no page faults were required (except those needed to initialize the buffer). At that point, the number of page faults increases roughly linearly with the size of the data structure, until the height of the tree increases at operation 2786. It then rapidly jumps from 0.7 page faults per operation to 2.8 page faults, where it remains roughly level, until the index height again decreases. In this demonstration, the internal records of the index remain cached in the buffer pool, and access to leaf pages results in page faults.

Because the performance of the A-tree depends primarily on its height, the above experiment could have inserted many more records with little additional degradation in performance. As long as the height of the A-tree does not change, the number of page faults per record access remains roughly constant. The number of records in a "full" tree is:

$$n = \left\lfloor \frac{yp}{r_i} \right\rfloor^{h-1} \left\lfloor \frac{yp}{r_l} \right\rfloor$$

where:

n is the number of leaf records in a "full" tree,
y is the page utilization,
h is the height of the tree,
p is the page size,
r_i is the size of an internal record,
r_l is the size of a leaf record.

Using the parameters of the above experiment, $y = .69$, $p = 1024$, $r_i = r_l = 16$, and $h = 3$, we find that the number of leaf records could be increased from 5000 to approximately 86,000 with no further increase in the height of the tree, and consequently no significant degradation in performance.

The concludes our discussion of A-tree performance. The next two sections consider the use of A-trees to implement the case where individual entities participate in multiple orderings, and in hierarchical orderings.

5.9. Multiple Orderings

The ordered set of records in an ordered relation (or in the leaf pages of an A-tree) may contain actual data records. Alternatively, they may contain pointers to the data records (i.e. *TID*'s). While this latter alternative results in less performance (an extra disk access is required to retrieve record data given a record pointer), it allows us to define multiple orderings over a single set of data.

5.9.1. Multiple Orderings in Sorted Relations

In systems such as INGRES, a facility for providing multiple orderings is provided by "secondary indices." A relation may be sorted independently on many different keys. One of these orderings may be designated as "primary." The actual data records will be stored in this order (these records constitute the "base relation"). All other orderings are "secondary." Figure 5.7 shows a relation with three fields, f_1, f_2, and f_3, each a key in a different ordering. The data records themselves, in the base relation, are sorted on f_1. One secondary index contains the values for f_2, plus a pointer to the data record in the base relation. This index is then sorted on f_2. The third relation similarly provides an ordering based on the field f_3. By scanning the appropriate relation, the data records may be retrieved in the desired order. The key values for the secondary ordering are duplicated in the secondary index, as a performance optimization.

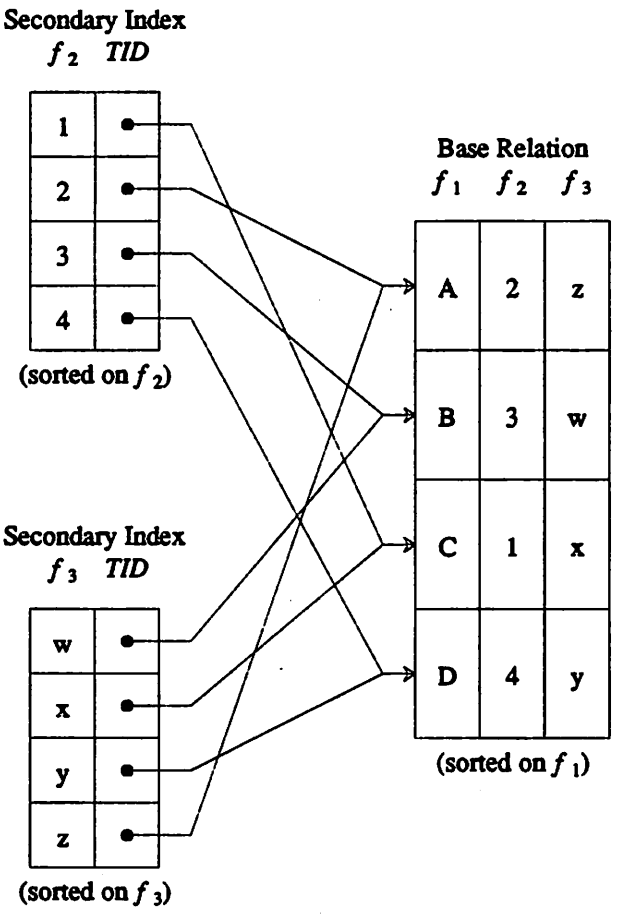


Figure 5.7. Multiple Sort Orderings

5.9.2. Multiple Orderings in Ordered Relations

This same technique may be applied to ordered relations, allowing entities to participate in multiple orderings. The syntax for specifying multiple orderings was presented in Chapter 3. Subsequent discussion will make use of the following example, which defines three orderings on relation *X*:

define entity X (*attributes of X...*)
define ordering (X)
define ordering A (X)
define ordering B (X)

The **define entity** statement results in an unordered relation. Each ordering specified by a **define ordering** statement results in an ordered relation. If the ordering is unnamed, as in the first example, the unordered relation associated with the child entity is converted into an ordered relation. The result-

ing ordered relation is primary. If the ordering is named, the resulting ordered relation is secondary (i.e. it contains pointers to the data records, which reside in another relation). This secondary relation is given the name of it its ordering. Thus, in the above example, the system creates three ordered relations. The relation *X* contains data records, and the relations *A* and *B* contain pointers to those data records.

A particular database system might also support inhomogenous orderings in this way, if it allows relations to contain records of different entity types (systems based on the "universal relation" model [MRW86, Men84, Sag83] typically allow this). Because such records are typically of various sizes, it is simpler to always implement them as secondary relations, even when unnamed (pointers to records of different types are presumably all the same size, and thus are easier to manage). For example, an ordering consisting of records from relations X and Y, intermixed, is specified by,

define ordering (X,Y)

and results in a secondary relation containing pointers to records in the X and Y relations. The system would provide a unique name for the resulting ordered relation, such as "X_Y" for the above example.

Having defined multiple orderings of a relation, the system must determine what attributes to copy from the base relation into the secondary relations. This depends on the ordered aggregates that are defined on the relations. Those attributes involved in the calculation of an ordered aggregate are to be included. For example, suppose the *X* relation contains the attribute "amount," and we wish to maintain a running sum, given ordering B:

define entity X (amount = **integer**)
define ordering B (X)
range of b **is** B
define inheritance b (balance = ordered_sum(b.amount))

This indicates to the system that,

(1) The attribute "amount" is to be duplicated from *X* to *B*.

(2) The ordered aggregate attribute "balance" is to be maintained over the *B* ordering. The base relation automatically inherits this attribute, so we may refer to "balance" as an attribute of the *X* relation.

We now consider a musical example. Chords in the music database are involved in multiple orderings. Every chord is independently ordered with other chords into a group within a voice, and with other chords under a common sync. Suppose we wish to maintain count information over each of these, so as to be able to refer to "the *n*'th chord in a given group" or "the *y*'th chord in a given sync". The base relation is called CHORD, and the two secondary indices are GROUP_ORDER and SYNC_ORDER. The indices are modified to type A-tree, each containing ordered_count aggregates to provide the ordinate information required.

When the **define inheritance** command is executed on a secondary relation, the system associates the inherited attributes (in the *attribute* system catalog) with both the primary and secondary relations. In this example, the CHORD relation inherits two new virtual attributes, group_ordinate and sync_ordinate. Here are commands that generate these orderings:

```
define entity CHORD (chord attributes...)

define ordering GROUP_ORDER (CHORD)
define ordering SYNC_ORDER (CHORD)

range of g is GROUP_ORDER
define inheritance g (group_ordinate = ordered_count(g))

range of s is SYNC_ORDER
define inheritance s (sync_ordinate = ordered_count(s))

modify GROUP_ORDER to A-tree
modify SYNC_ORDER to A-tree
```

The append command, when adding records, must now update these secondary indices also. The syntax of the **append** command must be further extended to manage this properly. For example, suppose we wish to insert a chord so that it precedes the second chord in its group, and follows the first chord in its sync. The command would be:

```
range of c1,c2 is CHORD

append to CHORD
    before c1 in GROUP_ORDER
    after c2 in SYNC_ORDER
    (attributes of new chord)
    where c1.group_ordinate = 2
    and c2.sync_ordinate = 1
```

One **before** or **after** clause is needed for each ordering that exists on the base relation. In the case that the user does not specify such a clause (or leaves out one of many), the system must arbitrarily place the object in the ordering. This is similar to the case where an insertion is made into a sorted relation where the inserted record was not assigned a key value. The default location in the ordering could be set (arbitrarily) before the first record. This would be similar to the current treatment of null values in the INGRES system.

5.10. Hierarchical Ordering

Up to this point, we have only considered global orderings, where the entire set of entities in a relation participate in a single ordering. We now discuss hierarchical ordering, where each entity in a relation has a "parent" (typically an entity in another relation), and an ordering applies to those entities that share a common parent.

The previous example of chords ordered within groups and syncs was simplified in its presentation, so as to ignore the hierarchical aspect of its orderings. In fact, two chords are only comparable under GROUP_ORDER if they are members of the same group (e.g. one cannot say a chord is before or after another chord according to GROUP_ORDER, if the chords are not in the same group; they are simply not comparable). This also holds true for the ordering of chords under syncs represented in SYNC_ORDER.

The hierarchical ordering in this example would be specified using the **under** clause, as we have already seen.

define entity GROUP (*group attributes...*)
define entity GROUP (*sync attributes...*)
define entity CHORD (*chord attributes...*)

define ordering GROUP_ORDER (CHORD) **under** GROUP
define ordering SYNC_ORDER (CHORD) **under** SYNC

range of g **is** GROUP_ORDER
define inheritance g (group_ordinate = ordered_count(g))
range of s **is** SYNC_ORDER
define inheritance s (sync_ordinate = ordered_count(s))

In order to support hierarchical ordering, we must extend our implementation of ordered relations and A-trees, as well as the syntax for **append** and **replace**.

5.10.1. Extending Ordered Relations to Support Hierarchical Ordering

The **define entity** statements in the above example create the GROUP, SYNC and CHORD relations. The **define ordering** statements generate the GROUP_ORDER and SYNC_ORDER secondary relations. An example of the relations resulting from the above definition are shown in figure 5.8. At the top of this figure is the musical fragment to be modeled, consisting of five chords, three syncs, and two groups. For every chord in the CHORD relation, there is a record in the GROUP_ORDER relation pointing to that chord. Additionally, records in the GROUP_ORDER relation contain a pointer to the GROUP record under which the chord is ordered. In the example of figure 5.8, three chords are ordered under group n in the GROUP relation, and two chords are ordered under group q. The type of the GROUP and CHORD relations are not defined here (they are presumably unsorted). The

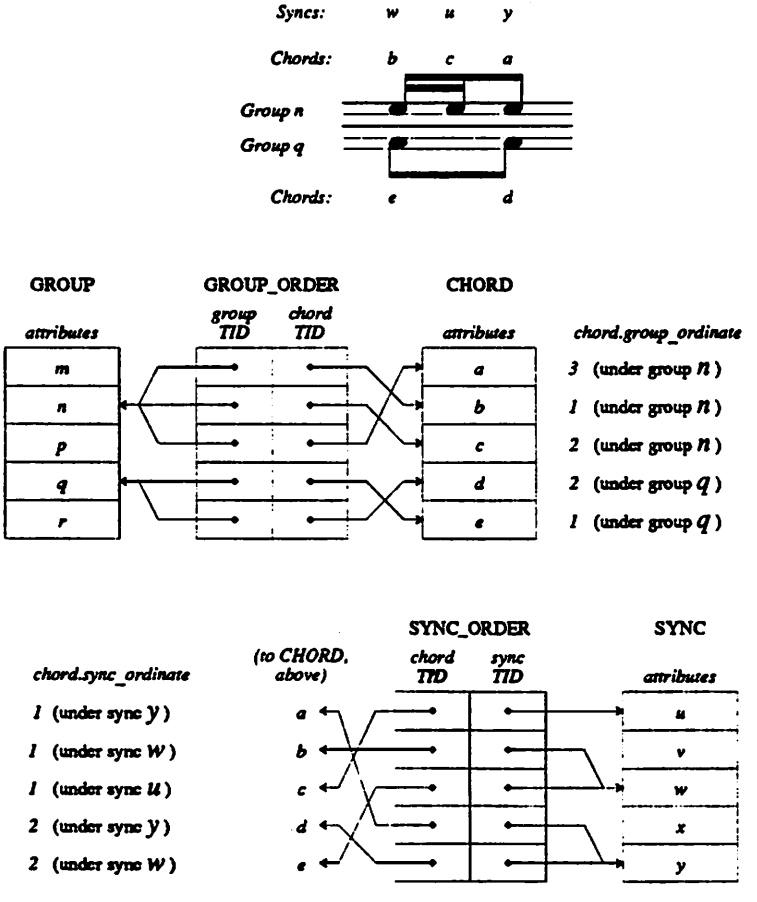


Figure 5.8. Hierarchically Ordered Relations

GROUP_ORDER relation is a *hierarchically* ordered relation, partitioned by GROUP TID. This partitioning is easily accomplished, for example, by sorting the ordered relation on its parent pointer field. Then, within each partition, the order of records reflects the ordering of the entities within that partition.

The SYNC_ORDER relation is ordered in the same way, with the syncs as parent entities. The partitioning and ordering of chords under syncs is independent of their partitioning and ordering under groups.

5.10.2. Insertion and Update of Hierarchically Ordered Relations

The syntax for append and replace statements must now be extended to allow records to be placed into relations that have ordered indexes. We therefore introduce the **under** clause. The following example inserts a new chord before the second chord in group q, and after the first chord in sync u:

```
range of c1,c2 is CHORD
range of g is GROUP
range of s is SYNC

append to CHORD
    before c1 under g in GROUP_ORDER
    after c2 under s in SYNC_ORDER
    (attributes of new chord)
    where c1.group_ordinate = 2
    and c2.sync_ordinate = 1
    and g is q
    and s is u
```

This statement results in the insertion of one record into each of the CHORD, GROUP_ORDER, and SYNC_ORDER relations. The record inserted into the CHORD relation would contain the attribute values for the new chord. The record inserted into the GROUP_ORDER relation is placed before the thrid record under group q. This record points to the chord entity in the CHORD relation, and has q as its *group TID*. Similarly, a record is inserted into the SYNC_ORDER relation before the tenth chord under sync u. This record has the TID of the new CHORD record as its *chord TID*, and u as its *sync TID*.

Suppose, in the above query, the **under** clause were omitted. The resulting query,

```
append to CHORD
    before c1 in GROUP_ORDER
    after c2 in SYNC_ORDER
    (attributes of new chord)
    where c1.group_ordinate = 2
    and c2.sync_ordinate = 1
```

would cause several chord entities to be inserted. The query would first determine all the chords that are second in their group and first in their sync (there would in general be several such chords, although the example in figure 5.8 contains only one), and insert a new chord next to each of them.

In this example, the parent TID of the inserted record is determined implicitly. In general, when a record x is inserted at a position before or after another record y, the parent of x is set to the parent of y. If the user specifies a conflicting parent (using the **under** clause) then the update is malformed. An example of such a non-functional update would be (referring again to figure 5.8):

```
append to CHORD
    before c1 under g in GROUP_ORDER
    (attributes of new chord)
    where c1 is b
    and g is q
```

Because chord b is not in group q, this command would be rejected by the system as malformed.

Replace statements are extended in the same way as **append** statements, with the **before**, **after**, and **under** clauses. When a replace statement is specified this way, it is treated as a deletion followed by an insertion, as before.

Although the above examples have all made use of secondary relations as hierarchically ordered relations, a base relation itself may be hierarchically ordered. For example, if we extend the TEXT relation to contain lines from several documents, rather just one, then the lines of TEXT are ordered under a given document. The TEXT relation is thus a hierarchically ordered base relation. The statements that specify this are:

```
define entity DOCUMENT (title = string)
define entity TEXT (line = string)

define ordering (TEXT) under DOCUMENT

range of t is TEXT
define inheritance t (line_number = ordered_count(t))
```

As shown in figure 5.9, the data records of the TEXT relation itself are partitioned by document, and

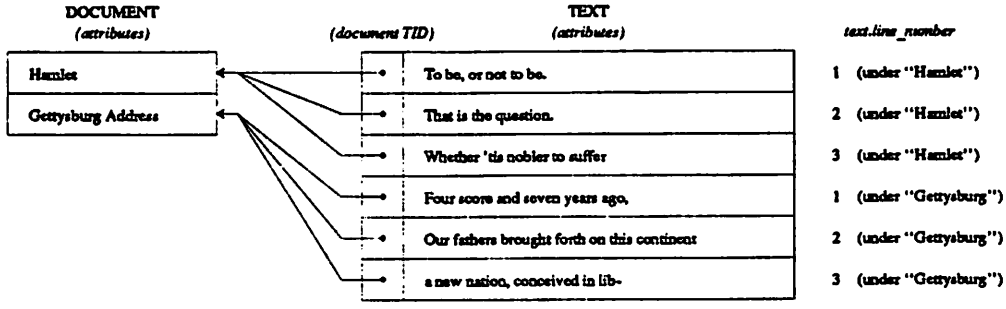


Figure 5.9. A Hierarchically Ordered Base Relation

ordered within each partition. Each record of the TEXT relation contains, in addition to the attributes specified in the **define entity** statement, a pointer to a DOCUMENT record. This ordered base relation may then be manipulated just as a secondary relation, with the ordering left unnamed. For example, to insert a new line after the second line of the "Gettysburg Address," the following statement is executed:

```
range of t is TEXT
range of d is DOCUMENT

append to TEXT
    after t under d
    (line = "a new line of text")
    where d.title = "Gettysburg Address"
    and t.line_number = 2
```

Although we could have explicitly named the ordered relation by saying "**after t under d in** TEXT," it is not necessary because the base relation is assumed when the **in** clause is missing, as in this example. Otherwise, the syntax of this example is the same as that for orderings represented by secondary relations.

5.10.3. Extending A-trees for Hierarchically Ordered Relations

In order to use A-trees to access hierarchically ordered relations, the search and scan algorithms presented previously must be modified to accommodate partitions.

The internal record of an A-tree over a hierarchically ordered relation contains summary information for each ordered aggregate, as before, but also contains the partition value (i.e. the parent TID)

for the last leaf under the subtree covered by that internal record. This is shown in figure 5.10. Given the specification of the TEXT relation, above, we have executed the statement,

modify TEXT to A-tree

to create this A-tree. The algorithm to update the summary information in the internal records is modified as follows to generate the internal record values shown in this figure.

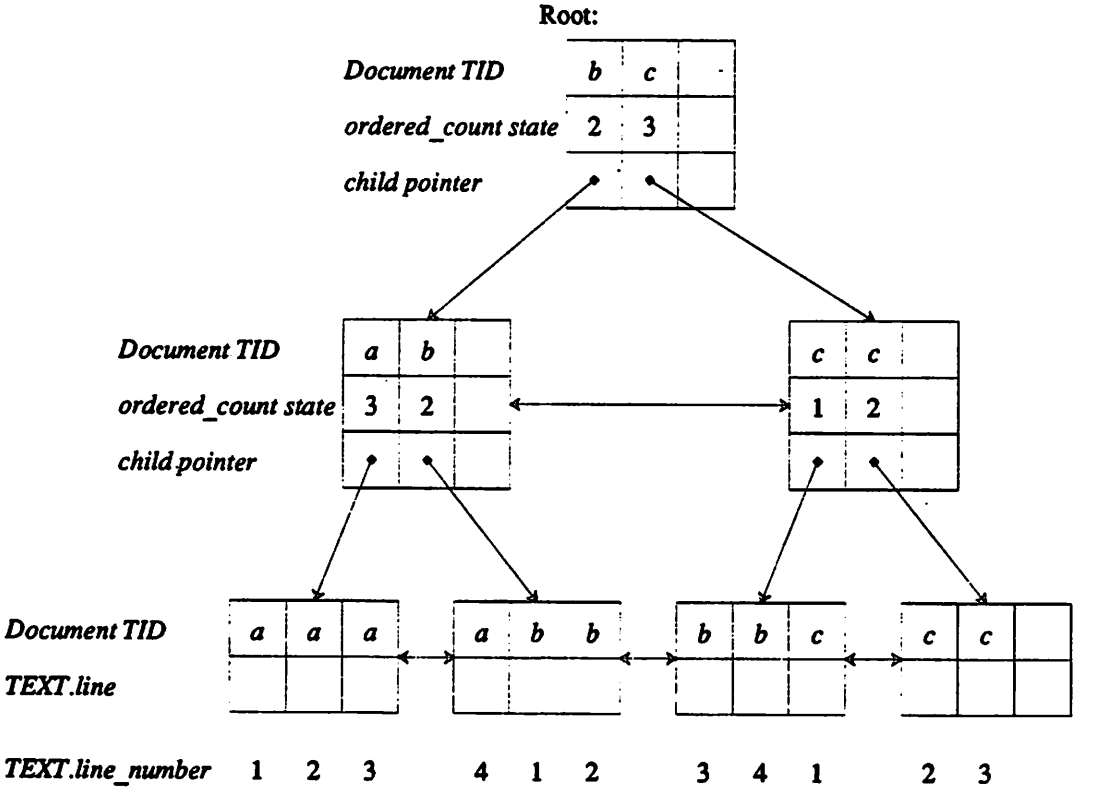


Figure 5.10. An A-tree for Hierarchical Ordering

Updating Summary Information

s is the current state,
p is the current disk page,
r is the current record,
x is the current partition

1. $p \leftarrow p'$
2. If *p* is the root, then stop
3. For each record *r* in *p*:
4. If *r* begins a partition, then
5. $s \leftarrow$*InitializeScan*()
6. $x \leftarrow$ partition of *r*
7. If *p* is an internal page, $s \leftarrow$ *NextInner*(*s*,*r*)
8. If *p* is a leaf page, $s \leftarrow$ *NextLeaf*(*s*,*r*)
9. $r \leftarrow$ the parent record of *p*
10. Set the "summary information" of *r* to *s*
11. Set the "partition" of *r* to *x*
12. $p \leftarrow$ page on which *r* lies
13. Go to step 2

In this algorithm, the scan is initialized at every partition, in steps 5-6, rather than once at the beginning.

All of the traversal algorithms are similarly modified. In each case where we scan the records of a page, we notice if the partition value has changed. When this occurs, we update the current partition value and call *InitializeScan()*.

The processing of aggregates that are defined as **ascending** or **descending** must be modified, because such aggregate values are now only monotonic within a partition, rather than across the entire relation. This can be seen in figure 5.11, where the line numbers (which are normally **ascending**) do not steadily increase over the TEXT relation. Thus, a command such as,

```
range of t is TEXT

retrieve (t.line)
    where t.line_number < 10
```

returns the first ten lines of *every* document in the TEXT relation. Rather than terminating the scan prematurely when a line number greater than or equal to 10 is found, we must continue scanning subsequent partitions.

5.11. Storing Orderings as Linked Lists

When the data in a relation participates in more than one ordering, the issue is raised as to how the data itself should be ordered so that access via each ordering is efficiently performed.

The simplest approach to this problem, and the one taken in the preceding discussion, is to select one of the orderings as primary, and store the data records of the relation in that order. Any access of the records via this primary ordering will thereby incur a minimum amount of paging activity. Of course, sequential access via any secondary ordering will cause considerable paging activity, since records that are "near" each other in this ordering are not necessarily nearby each other in the primary ordering (where the data associated with the records is kept). We might call this approach "order by placement". Whether an object is before or after another object depends on their respective storage locations. Figure 5.11 gives an example of this type of ordering. As we have seen, we can retrieve the records of such a relation in a particular order by selecting the ordered heap that implements the desired ordering (either the primary or one of the secondary relations), and scanning it sequentially. We have also seen that an A-tree may be built over each ordered heap to eliminate the need for sequential scans in many cases.

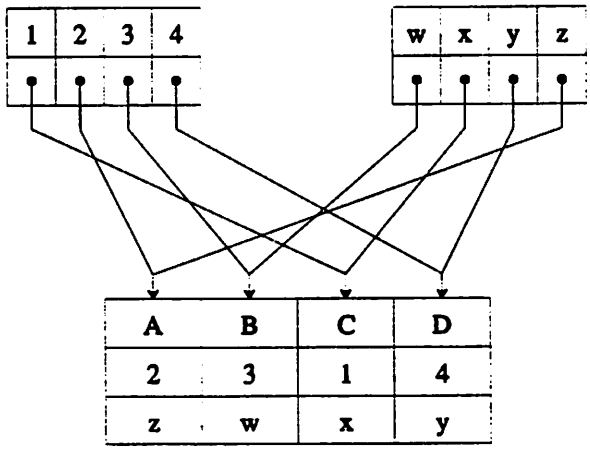


Figure 5.11. Representing Order by Placement

5.11.1. The Linked Heap Structure

An alternative implementation would be to represent the orderings by actual pointers from a data record to each of its successors records (a record has one successor per ordering in which the record participates). An example of "order by pointers" is shown in figure 5.12. The records in this implementation are organized into linked lists. This data structure will be called a "linked heap." We first discuss the structure of the linked heap, and its integration with A-trees; we then compare its advantages and disadvantages with those of ordered heaps.

The placement of the records onto pages may be arbitrary, since it is no longer a factor in determining what the "next" record is. In order to retrieve the records of this relation in a particular order, we start at the first record in that ordering (whose address must be maintained separately by the system), and follow the successor pointers for that ordering to retrieve successive records.

When using linked heaps, the **define ordering** statement, rather than defining a new ordered heap, defines a single additional attribute on the base relation (generated by the **define entity**

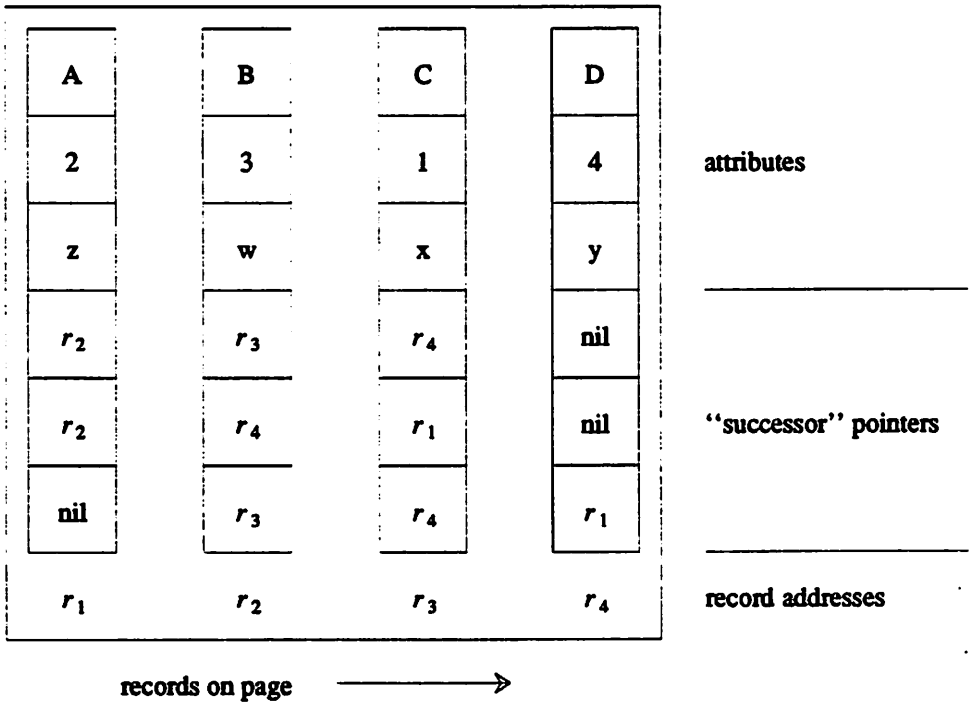


Figure 5.12. Representing Order by Pointers

statement), whose value is the successor pointer for each record in the ordering.

When inserting a record into a linked heap at a particular point in the ordering, we may place the new record into any free slot on existing pages, or at the beginning of a new page, and adjust its link pointer and that of its predecessor so that it is properly incorporated into the chain of records.

By slightly modifying the A-tree structure, we can provide efficient access to ordered aggregate values on multi-ordered relations stored in linked heaps. When using secondary relations, every A-tree had its own ordered heap at the leaf level. Now, because all ordering information is encapsulated in the single relation, several A-trees will share this relation for their leaf levels. This is shown in figure 5.13. In this example, the internal records contain the summary information to maintain the ordered_count aggregate over the records in the linked heap. The A-tree structure is modified as follows:

The leaf level of the tree is a linked heap rather than an ordered heap.

Each record in the linked heap contains one "successor" pointer per ordering. When a single entity participates in multiple orderings, several A-trees share the same linked heap for their leaf level, although each will use a distinct set of successor pointers.

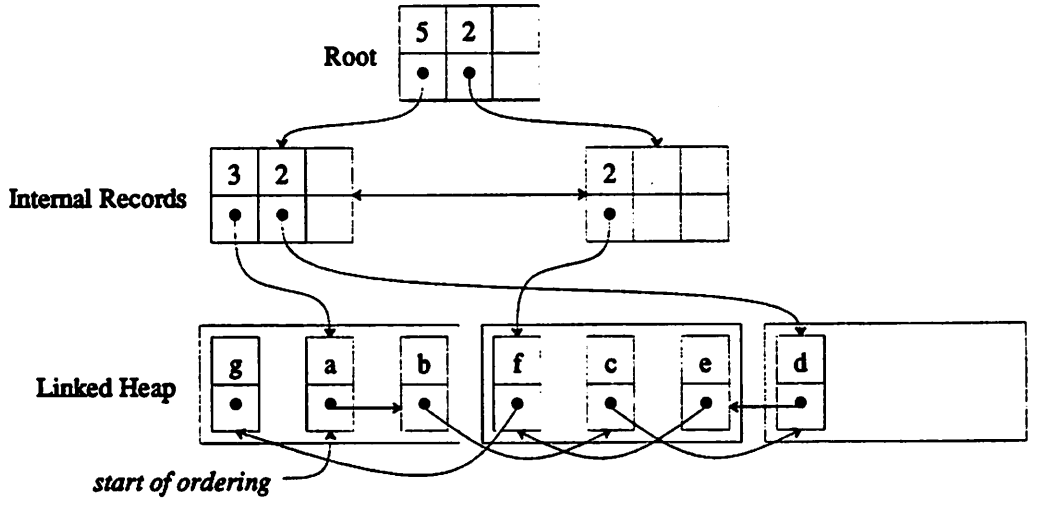


Figure 5.13. Building A-trees Over Linked Heaps

Internal records at the lowest inner level of the A-tree contain pointers to leaf records in the linked heap, rather than pointers to pages as before. The downward traversal will scan sequentially through internal pages, and follow links at the leaf level. The set of records at the leaf level that are linked under a single internal record will be a called a *chain*. In figure 5.13, the first chain contains records {a, b, c}, the second chain is {d, e}, and the third is {f, g}.

To find the record possessing a particular ordered aggregate value, we select the A-tree associated with the appropriate ordering and traverse down to the leaf level as before. At the leaf level, rather than performing a page scan, we traverse the chain whose first element was indicated by the lowest internal record.

The maximum number of leaf records directly under a given internal record was previously constrained by the size of a disk page. Now, this restriction is removed. Thus, there is no necessary upper bound on chain length. When inserting a record into the linked heap, the system is free to split the chain or not. If we decide not to split, the chain gets longer. If the chain is split in two, an internal record is inserted for the new chain.

There are two obvious approaches for determining when to split a chain. In the first approach the system (or perhaps the user) fixes an arbitrary maximum chain length. As records are inserted, the height of the tree may increase, but the length of chains will be bounded (this is similar the behavior of B-trees). Another possibility is to fix the height of the tree, thus fixing the total number of chains. As insertions are performed, the chains get longer (this models the behavior of ISAM indices).

5.11.2. Comparing the Two Approaches

The use of linked heaps is arguably more complex than our previous proposal. The disadvantages associated with complexity apply in this case: the resulting system is harder to implement, debug, and maintain.

Since A-tree traversal uses two different scanning mechanisms, sequential scans of the internal pages, and link traversal through the leaf records, the implementation of these tasks is less modular and more cumbersome.

Linked heaps may be more easily corrupted than ordered heaps. If a link field is destroyed, traversal of the heap is impossible. In an ordered heap, there is no data corruption (except perhaps to the double links between leaf pages) that can prevent successful traversal of the relation.

There are two advantages to this new storage strategy. First, it may be space efficient, since the attribute parameter values need no longer be copied into each secondary index. The space required for TID pointers in the secondary index is balanced by the space required for successor pointers in the linked heap. Some additional space may be required in the internal records of the A-tree, since the lowest level contains record pointers rather than page pointers. This space advantage only occurs if there are multiple orderings, since a primary ordered heap contains no TID pointers and thus takes less space than a single linked heap.

More importantly, because the placement of records on pages is now arbitrary, we are free to organize the records so that for a given record, many of its successors share its disk page, rather than just one. In other words, data can now be effectively clustered. In the previous proposal, sequential scan of the primary relation was very fast, but on the secondary relations was much slower, because for every record in the secondary relation, we incur a page fault to get the actual data from the primary relation. Using linked heaps, the data may be clustered with respect to all orderings, rather than only a single (primary) one. If the clustering is successful, we may traverse all of the orderings with moderate performance.

5.11.3. Clustering

A linked heap with n records ($n>1$) and p orderings constitutes a graph G with the following characteristics:

- every record in the linked heap is a vertex v of the graph, each vertex has a weight w_v equal to the size of the record.
- every successor pointer from a record to another record is an edge e in the graph (we will consider the edges to be undirected), and each edge has a weight w_e whose value is determined below.

- The graph contains $p(n-1)$ edges, and the degree of every vertex is between p and $2p$

The graph for the linked heap shown in figure 5.12 is presented in figure 5.14. For clarity, the edges from each of the three orderings are shown using a different line type: solid, dotted, and dashed.

For a given application, we may have information as to the relative frequency with which different edges of the graph are traversed. Each edge is assigned an edge weight, proportional to this frequency. In the absence of this information, we can give every edge equal weight. The problem of clustering the linked heap so as to minimize disk paging activity now reduces to a classical problem called the "graph partioning problem." That is:

> Given a graph G with weighted vertices and edges, partition the vertex set such that the total vertex weight of every partition is less than p, and the total weight of all edges whose end points lie in different partitions is minimized.

The first constraint on vertex weight corresponds to the requirement that all the records in a partition must fit onto one disk page. The latter constraint minimizes the frequency of traversing from one disk page to another.

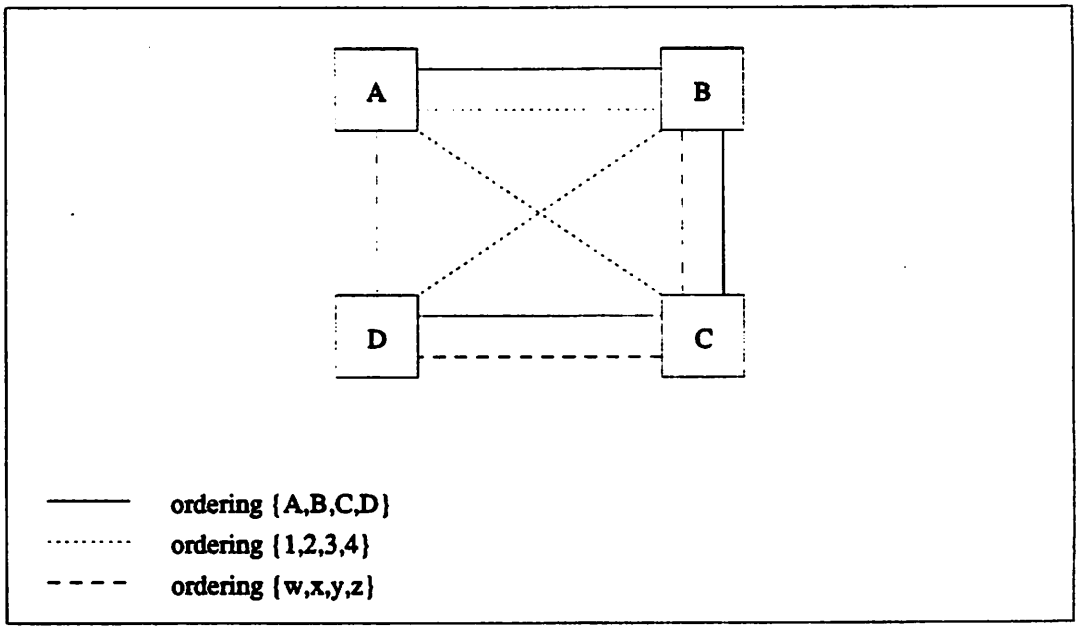


Figure 5.14. Graph Representation of a Linked Heap

There currently exists a large body of research on graph partitioning that may be brought to bear on how such data should be clustered. Although the optimization problem is NP-complete [Mac78], there are particular types of graphs, such as planar graphs [Had75] and trees [KuM77, Luk74], for which fast algorithms are known. Unfortunately, multi-ordered lists (for example, the list represented in figure 5.14) are not of either of these forms.

5.11.4. Clustering Experiments

Another approach has been to find results that are good, though not necessarily optimal. Such algorithms use heuristics to determine how a graph should be clustered [AJM84, FiM82, KeL70, Luk75, Mac78, RSS84]. These algorithms are typically tested empirically, that is, by applying them to either random graphs, or graphs that arise in a particular application. Some previous empirical experiments in graph clustering have focused on actual CAD circuits [FiM82]. Such a circuit consists of modules with various interconnections, and the clustering problem is to divide up the modules into groups (e.g. areas of a chip, or a circuit board) while minimizing the number of interconnections between groups.

In order to have a rough idea as to whether such heuristic algorithms would be useful for multiply ordered lists, we performed some simple experiments. In them, we compare two clustering algorithms:

Primary ordering: One ordering is selected as primary, and the records of the relation are assigned to pages according to this ordering. This is the same clustering that is performed when using an ordered base relation with secondary indices.

Iterative Min-Cut: This algorithm has been suggested for partitioning CAD circuits [KeL70]. We use a linear time version of the algorithm presented in [FiM82]. The algorithm basic idea of the algorithm is presented here. A heuristic is used to partition the entire graph into two halves such that the number of edges that cross between the two halves (i.e. the cut set) is small. Each half is further partitioned using the same heuristic, and this continues recursively until the number of elements in a partition can fit on one page.

In order to find a good division of the graph into two partitions, the graph is first partitioned arbitrarily. Then one node at a time is moved from one partition to the other, so as 1) to keep the partitions

of roughly equal size, and 2) to decrease the size of the cut set, if possible. Even if no move decreases the cut set size, the best (i.e. least detrimental) move is performed anyway, in order to "climb out of local minima." After a node is moved, it is "frozen" and not subject to being moved again. After every node is frozen, the partition seen so far that has the smallest cut set is used as the start of another pass. When a complete pass cannot make the cut set any smaller, the algorithm terminates.

The multi-ordered list on which these algorithms were tested had the following characteristics:

- the list contains 10,000 records,
- each record participates in 3 orderings,
- the maximum page size is 64 records.

We hypothesized that the correlation between the orderings would strongly affect the results, so we generated the second and third orderings so as to have particular correlation coefficients with respect to the first ordering. Rather than selecting the successor of a given record randomly in the secondary orderings, a small set of successor candidates was determined by considering those records "near" to the given record in the primary ordering. For example, by setting the "nearness" parameter to 10, the successor for a given record in either secondary ordering is drawn from the set of 10 records nearest to the given record in the primary ordering (five before and five after). The experiment was tested at several different values for the "nearness" factor.

Our results are summarized in figure 5.15. For each clustering algorithm, the quality of the clustering is plotted against the correlation among the orderings. The y-axis represents the percentage of pointers which cross a disk page boundary (when such a pointer is traversed, a page fault is generated). The x-axis represents the "nearness" factor. At the left edge is perfect correlation (all three orderings are the same), and at the right edge is no correlation (selecting from 10,000 neighbors in a universe of 10,000 records is identical to selecting a record at random).

As this diagram shows, selecting a primary ordering is always superior to the min-cut algorithm, seeming to indicate that sophisticated graph clustering is not a useful tool to optimize access to multiply ordered lists. This seems intuitively plausible, because multiple orderings display a particular pattern of edges (long linked lists) for which a primary ordering is reasonably defined. A "primary" ordering is not necessarily determinable in an arbitrary graph. The min-cut algorithm makes no

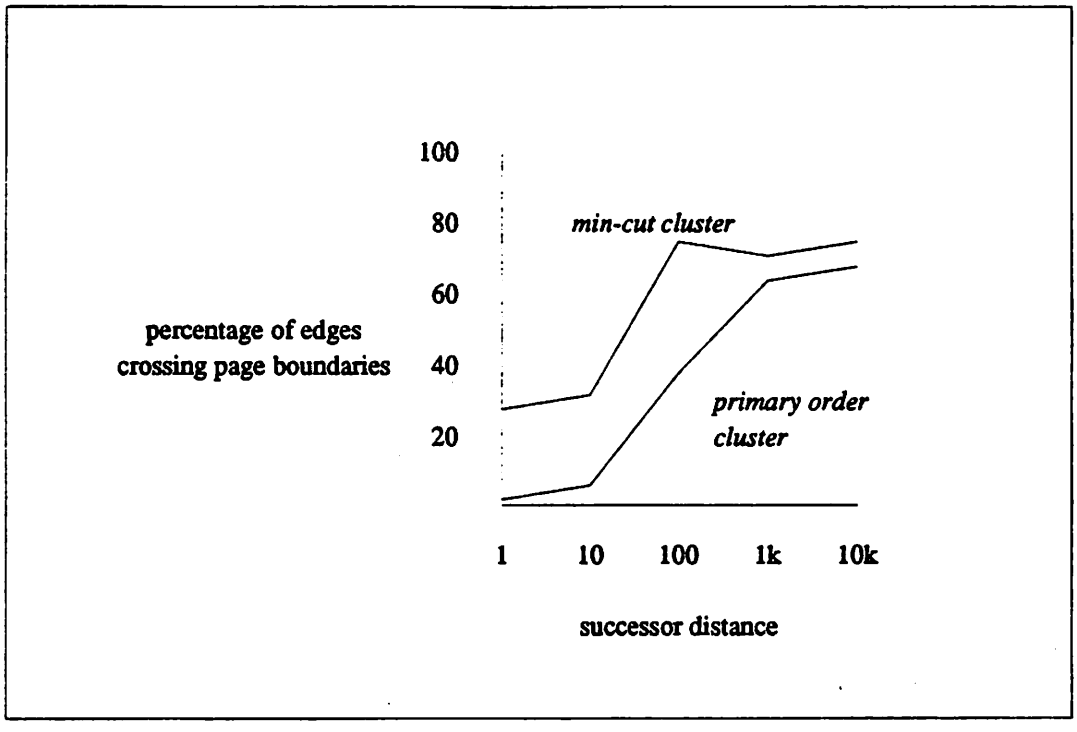


Figure 5.15. Comparison of Min-Cut Clustering with Primary Ordering

assumptions about the structure of the graph and, while more broadly applicable, performs less well. In any case, certain questions remain open:

There may be other clustering algorithms that perform better on multiply ordered lists. Algorithms such as simulated annealing [AJM84, RSS84] might be useful, especially if the number of entities and the number of orderings are both large.

Because there is no global order among hierarchically ordered records, it is not clear how the records should be ordered on the disk (this is one case where a primary ordering is not readily determinable, as mentioned above). Although the records within a partition may be ordered, it remains unspecified how the partitions as a whole should be ordered. Does it make a difference whether there are few large partitions (approximating a global ordering) rather than many small partitions (with fewer sibling pointers)?

In hierarchically ordered data, there are pointers not only to successors, but also to parents. If the system permits children and parents to be clustered together, the resulting graph structures are less regular, and perhaps primary ordering would be less appropriate under those circumstances.

Clearly, there remain a large number of open issues associated with the clustering of graph data. At least preliminarily, it appears that primary ordering is a reasonable clustering technique for both single orderings and multiple orderings.

5.12. Additional Issues

This section will consider a number of other issues that have surfaced in the course of the development of the A-tree proposal. They are concerned with various ways in which the current implementation could be made more general or more efficient. In general, the issues have been clearly identified, and their solution remains open for future research.

5.12.1. User Access to Tuning Parameters

We have identified certain parameters whose values might be determined by the user. The effect of various settings of these parameters on database performance remains to be explored.

When an A-tree is initially built (with the **modify to A-tree** command), the initial utilization of leaf pages may be set arbitrarily. As we have seen, with repeated insertions and deletions, this utilization should stabilize at 69%. A user may wish to set the initial utilization much lower, if a large number of insertions are to be performed shortly after the A-tree is built. The leaf pages will be relatively empty, and less likely to overflow when the insertions are performed.

If ordering is implemented using linked heaps, then the issue of determining chain length is still open. A user may wish to decide that the length of chains is to be fixed, and the height of the index is variable. This follows the style of B-trees, and provides stable performance in the presence of a large number of insertions and deletions. An empirical study for a given application could determine an optimal value for this chain length.

Alternatively, the user may prefer that the height of the tree is to be fixed, and therefore the number of chains is fixed, while the chain length is variable. This is similar to ISAM indexes. Because the internal pages of the A-tree will not be modified, they can be utilized fully (we need not

reserve empty spaces for additional internal records). This provides optimal tree height and index size. However, chains may become arbitrarily long, and thus the index performance degrades with insertions and deletions.

5.12.2. More Efficient Tree Traversal

Each internal record contains information that summarizes the contents of the data records in the leaves of its subtree. For top down search over ordered aggregates, every page on the path from root to leaf is scanned from left to right. For each internal record scanned, a call to a user routine (NextInner) is made to calculate a new cumulative state value.

It might be more efficient to precalculate the values of the state at each record of a page, and store this cumulative state value, rather than each individual summary record.

The advantage of storing the cumulative state value is that traversals which scan the internal pages of the index may proceed with no calls to the NextInner routine. If calls to user supplied procedures are expensive (in a particular system, such calls might incur a context switch within the operating system, additional time to demand load the user code, or overhead associated with some form of remote procedure call), this will provide a large savings.

The disadvantage of this approach is that the cost of updates is increased, and one additional function must be supplied by the user for each ordered aggregate. According to the algorithm to update summary information given previously, any time a leaf page is modified, every page on the path from leaf up to (but not including) the root is scanned. The resultant state from the scanned page is stored in its parent's internal record, r_p. With this new proposal, the cumulative state of every record after r_p in the parent page must also be modified. The cumulative state to be stored in r_p may be determined by calling NextInner with the cumulative state stored in the record before r_p, and the summary information resulting from the scan of the child page. To determine the new cumulative state to be placed in the next internal record (call it r_q), the user must provide a new function called UpdateState.

The UpdateState function takes both the old and new cumulative state of r_p (the previous record), and the old cumulative state of r_q (the current record), and determines the new cumulative state for r_q. Such a function typically involves inverting the function of NextInner (i.e. given the old

r_p and r_q values, we determine the summary information of r_q, which we add to the new cumulative state in r_p to get the new state for r_q).

Time must be spent scanning the root page in order to update the cumulative state for every record there. This scan was not required by the original proposal.

The inversion performed by UpdateState is not necessarily defined for all ordered aggregate functions. For example, the ordered_max function (whose value at a given record is the maximum of some attribute value over the set of records up to and including the given one) has the "current maximum" as the cumulative state at a given point. Given only the cumulative state at a given internal record and at its predecessor, there is no way to determine the summary information of its subtree: if the two cumulative states are equal, the maximum value may be in the subtree of r_q or in the subtree of its predecessor. In that case, the new cumulative state cannot properly be determined for r_q without accessing its child page. In no other case does a user-coded routine need to perform a disk page access, and the performance penalty to do so may be unacceptable.

In summary, this proposal makes retrievals faster, and updates more expensive. In an environment where retrieval is common but update is rare, this will provide enhanced performance. The range of implementable functions may be narrowed, if the new user-defined function *UpdateState* cannot be implemented.

5.12.3. Parent Pointers

The above discussion of A-trees assumes that every page of the data structure contains a pointer to its parent record (except, of course, the root page). This parent pointer is used in bottom-up traversal of the tree. Such a traversal would be required to perform a query such as:

```
retrieve (TEXT.line_number)
    where TEXT.line = "A line of text"
```

Some search (possibly via a secondary index) of the TEXT relation would find a record containing the appropriate line of text, and then a bottom-up traversal would determine that line's ordinate position, *line_number*.

In B-tree implementations where the leaf records are sorted on a key field, such parent pointers are unnecessary, since the path from leaf to root is easily determined by taking the key of a leaf record

and performing a top-down traversal based on that key. The page addresses at each level of the tree can then be saved for a subsequent bottom-up traversal.

This approach does not work when there is no key field stored in the leaf records. For example, in the OB-trees developed by Lynn [Lyn82], the absence of parent pointers disallows the retrieval shown in the previous example. It is not possible to determine the "line_number" of a given line of text, unless that line was found by performing a top-down traversal of the index.

Although this problem is solved by maintaining parent pointers, the cost of this feature may be quite high. In particular, any time an internal page is split or merged, a large set of child pages must have there parent pointers updated. Each such update requires a page access.

This issue can be resolved in three ways:

- Do not maintain parent pointers, and disallow accesses that require them.
- Support parent pointers on child pages, and tolerate the excess cost of maintaining them.
- Support parent pointers, but rather than keeping them on child pages, store them in a separate data structure.

This last alternative proves to be a desirable choice. If the size of such a structure is reasonable (and it will be shown that this is true), a relation can be maintained in memory that associates a parent record address with every internal page of the A-tree. If we hash this relation on the page address, we can efficiently determine the parent record for a given page.

How large is such a list? Using the parameters of our A-tree simulation, a page address requires 16 bytes, and a record address uses 24 bytes, so one entry in our list contains 40 bytes. One such entry is required for every internal record. Our simulation showed a full tree of height three (1.4 Megabytes of data) to contain 86,000 leaf records, and about 2000 internal records. Such a relation therefore requires a to have a parent pointer relation of less than 80K bytes.

In some systems, this may be small enough to fit permanently into main (or virtual) memory. Alternatively, since locality of reference by a user in the ordered relation will be reflected in the locality of reference to parent pointers, a memory cache of parent pointers might be effective. In such a scheme, recently used parent pointers would be buffered in memory. If a parent pointer was needed

but not available, the internal records of the A-tree could be scanned to find it. It would then be rapidly available as long as it is retained in the cache.

It should be noted that this internal structure need not be crash recoverable, since it can easily be reconstructed by traversing all the internal nodes of the A-tree. When the parent pointers are modified, such as when a child page is split, the parent pointer relation must be updated to reflect the current state of the database.

5.13. Summary

A-trees provide a general approach for supporting user-defined indexing structures over abstract data types. They are modeled after B-trees, with the information stored in the internal nodes of the tree placed directly under user control. By supplying a small number of routines a user can easily simulate existing ordered access methods, such as B-trees and OB-trees.

Additionally, the A-tree supports aggregate values calculated over ordered relations. Of particular interest are thoses aggregates, known as ordered aggregates, which supply for each record of a relation an aggregate value based on that record and all previous records in the ordering. Examples of this approach are found in databases that maintain ordinate record numbers and those that maintain a per record running total over an attribute of the relation.

It has been demonstrated that the performance of these A-trees under dynamic insertions and deletions is the same as for B-trees.

By including a method of partitioning the data stored in an A-tree, hierarchical orderings are supported. By permitting A-trees to be used as secondary indices, multi-orderings are supported.

In the case where multi-orderings are present, the data may be stored as a graph. This results in a space savings, and potentially a time savings as well. In this case, the issue of how the data (i.e. the nodes of the graph) should be clustered needs to be addressed. It remains to be seen whether general purpose graph partitioning algorithms may be brought to bear to solve this problem.

CHAPTER 6

Temporal Data Management

Information which represents music is fundamentally temporal in nature. This, of course, is not unique to the musical domain. Numerous other information domains, such as on-line calendars [And81], project scheduling [MoP64], transaction management [SLR76], (with its issues of temporal serializability [Pap79]) and version maintenance [KaL82] involve the manipulation of temporal data. It is surprising, therefore, that none of the three major data models, hierarchical, network, nor relational, as originally formulated, addresses the issue of time management.

A large amount of research has therefore gone into various extensions to these models to incorporate temporal data. A reading of the literature (see, for example, the survey by Bolour [BAD82]) makes it obvious that the issue of what constitutes an appropriate representation for this data is far from settled. In the course of this chapter, we discuss how time in the music database is different, though related, to many of the references to temporal data to be found in this research.

This chapter begins in section 6.1 by surveying a number of these research efforts, with an eye toward determining the characteristics of the musical information domain which distinguish it from other domains. Just as with the representation of musical data in general, we are concerned with the management of temporal data on two distinct levels. Section 6.2 will first focus on the conceptual level. In order to crystalize the semantics of time within the musical data manager, the notions of *time line* and *event* are defined. We make use of the hierarchically ordered entities of chapter 3 to model these entities. We define a set of queries over the data, and show in section 6.3 how A-trees provide for the efficient solution of these queries.

The actual query language programs that implement time lines using A-trees will then be given in section 6.4.

An interesting application of multiple time lines to music involves the mapping between different musical time frames, such as score time and performance time. We introduce the concept of *time maps* and *tempo maps* in section 6.5, and show how they can be used in conjunction with time

lines to provide a flexible mapping between time frames.

6.1. Time in Database Research

There seems to be agreement that the lack of time modeling facilities in first generation data managers was a serious deficiency (a discussion of this can be found in [ACJ83]). There does not appear to be a consensus, however, on how to address this problem. The major reason seems to be that there are actually three independent data management problems involving time (roughly in the order of the amount of attention they've received from the database research community):

- How to model data which is updated over time.

Most databases can be viewed in this way, since update itself is a temporal phenomenon. The data in these databases are not temporal in nature, except insofar as they model facts in their domain that are true over a period of time (i.e. until they are updated). Queries against these databases are typically of the form, "what is the value of datum x at time t?"

- how to model temporal data itself.

Examples of temporal data are processes, events, and calendars. The data in such databases is itself explicitly temporal in nature. We can ask a query such as "At what time does event e occur?" This data is subject to update just as in the previous case, and so we can combine the two perspectives to get queries of the form "As of time t_0, at what time does event e occur?"

- how to model the natural language constructs of tense and modality within the query language.

Facts in the database represented by natural language often have complex temporal aspects. A sentence such as "John had been going to the store." represents an event with temporal attributes that are not trivially modeled. Although it might be interesting to explore the application of temporal logic [FiC73, Pri67, ReU71] to musical databases, we will not pursue this issue here.

6.1.1. Historical Databases

Often, a database contains information intended to model some aspect of "the real world". A datum in such a database is an implicit assertion of a fact (e.g. the fact that a given employee earns a given salary). Database systems have been concerned with the "current view" of the relevant information. In other words, as the state of the world changes, the database is updated after the fact to

reflect these changes, and the previous state is "forgotten." Such a database which contained employees and their salaries could answer the query:

What is Jane's salary?

but could not answer the query:

What was Jane's salary on March 2, 1978?

A solution to this latter query is made possible by maintaining an *historical database*. Whenever an update to a relation is introduced into the historical database, rather than overwriting previous state information, the new information is appended to the relation. The previous state information is thus preserved. This non-destructive update was first suggested in [Sch77]. Proposed systems that incorporate this nondestructive update to maintain historical state information include GemStone [CoM84], TQuel [Sno84], and Postgres [StR85].

Clifford and Warren [ClW83] discuss a formal semantics, based on intensional logic, for this data model. In particular, they model the database as a succession of states; each state transition is triggered by an update to the database.

It was noted by Lum, *et al.*, in [LDE84] that this approach only supports a single notion of time, "physical time". It did not support updates which take effect in another time frame. For example, suppose the user wishes to form an update:

(At 5/1/86) raise Sue's salary to 60,000 dollars, retroactive to 4/1/86.

There is no way to record this fact. Its was suggested therefore that both physical time (i.e. the time that the update is processed) and logical time (i.e. the time that the update is "effective") be recorded. This allows queries to be posed that look like:

What was Sue's salary on 4/1/86 (logical time) as of 5/1/86 (physical time).

Snodgrass argues in [SnA85] that the distinction between physical time and logical time is not well defined, and that a more appropriate division of time distinguishes between *valid* time and *transaction* time. The valid time of a record indicates precisely the period during which the values in the record accurately model the real world. The transaction time of a record indicates the period from when a record is entered into the database until it is superceded by an updated version. These two time

lines are conceptually independent. In the same proposal, Snodgrass includes the notion of *user-defined* time for times related to a particular event. In the employee database, for example, the event "grant Sue a raise" has a physical time, at which the updated salary is recorded in the database, a valid time at which the raise is actually granted, and a user-defined time when the raise becomes effective.

It appears that these types of extensions could continue indefinitely; particular applications might refer to an arbitrary number of "interesting" time lines. Music, in particular, is often structured by using similar sequences of events, each organized within a different time frame. Thus, a canon or fugue consists of similar melodies shifted in time with respect to a particular point in time. Another example is found in compositions where two instances of a melody are started at the same time, but proceed at slightly different rates (so called, "phase music"). One may view such a composition as consisting of two identical instances of event sets, each performed in a different time frame. This notion will be made more precise in our discussion of time maps, in section 6.5.

6.1.2. Modeling Temporal Information

Existing research on modeling explicitly temporal information has focused on the notion of an *event*, which is the unit of temporal action.

The TERM (Time-extended Entity-Relationship Model) system [Klo83], is an example of a system which includes a notion of "valid time" in order to support histories of entities within the database. This system, however, also models *point events*, which are inherently temporal in nature. They demonstrate a number of different types of event sequences, for example:

- Events are points in time at which an attribute takes on a new value (e.g. via update). The value of the attribute remains constant until the next event changes the value of the attribute for the same entity.
- Events are points in time at which an attribute is known (e.g. via a measurement) to have a certain value. The value is continuously changing over time. It is known with precision at event-points, and perhaps can be determined at other points via an induction formula, such as interpolation.

- Events can take place at regular intervals along a time line that does not match real ("clock") time. For example, banking deposits and withdrawals occur every banking day. No meaning is ascribed to the value of an attribute in between these times.

Shoshani and Kawagoe [ShK86] continue this analysis. They define "time sequences", as values of an attribute associated with points in time for a given entity. These time series are classified according to their structure and interpretation as being:

- regular or irregular in the time domain,
- continuous, step-wise constant, or discrete in the interval between points. A special case of the step-wise constant is where the attribute value is boolean. This constitutes an *interval* (from attribute true to attribute false).

Similar event models were developed earlier by Bruce [Bru72] and by Findler [FiC73]. These proposals were oriented toward a natural language interface that managed temporal queries, particular the mapping of tense in English to precise query language constructs. Findler was additionally concerned with *temporal inference*, inferring the overall temporal order of events given a number of specifications such as "event e_1 preceded event e_2".

Various proposals for organizing temporal information within the conceptual level of a database schema have been developed. Anderson [And81] developed a model that organizes events into decomposable hierarchies, known as *processes*. Barbic [BaP85], in the Temporal Semantics Office System, considers the relationship between events especially important, and introduces into his model the notion of *causality* as a connection between events.

6.1.3. Modeling Musical Events

The role of temporal data management in the music domain has been recognized explicitly by a few researchers. Some tools have been developed for editing sets of events, such as the ELED system [DeK85]. Other musical score editors, such as INTERSCORE [Pru84b] have made use of the event typologies mentioned above. The INTERSCORE system maintains three semi-independent time lines under user control:

- a conceptual representation of a composition, which would include both static and dynamic tempo indications (e.g. metronome markings, *accelerandi*, etc.),
- the performance of the composition, which would include *rubato* and phrasing,
- a "piano roll" score that the user may edit.

An important issue in timing musical events is that of synchronization. Groups of events are often specified to occur simultaneously. Although the temporal location of these events might be ambiguously specified, the simultaneity of those events must not be violated. One construct used to model flexible temporal location of events while preserving simultaneity is the *time map* discussed later in this chapter.

6.2. Events as Ordinate Data

We can define our temporal domain to consist of two types of entities, *time lines* and *events*. In this section, we specify the attributes of these entities, and explore a set of operations specific to time lines.

An event is the atomic unit of activity. It is defined as follows:

```
define entity EVENT
    (start_time = integer,
    duration = integer)
```

We define a *time line* to be the time over which a particular set of musical events occur. A particular composition may consist of a number of time lines. A time line is defined as:

```
define entity TIMELINE
    (duration = integer)
```

The single attribute, "duration", determines the total duration of the time line. A typical value for the duration of a time line might be the temporal distance from the beginning of the first event in the time line to the end of the last event.

Events are hierarchically ordered under time lines:

```
define ordering (EVENT)
    under TIMELINE
```

Having characterized these time lines, consider how we might want to edit them. The standard update mechanisms (allowing, for example, update of any attribute of any tuple) can be applied to any of the tuples in our relation to give simple, standard editing functions. For example, changing the duration of event merely involves replacing the value of a "duration" attribute in an instance of the EVENT entity.

However, there are more complex editing operations which are particular to time-ordered data. Operations that modify time lines as a whole are found in most systems that implement the musical task of "sequencing" (organizing and editing the order and placement of events). We will describe to types of insertion, "splice-in" and "overlay", two types of deletion, "splice-out" and "remove", and a retrieval operation, "get-event".

6.2.1. The Splice-in Operation

This operation makes a "break" in the first time line at the *insertion point*, and "splices" the second time line into the break. All events in the first time line which are initiated after the break get slid forward in time by the duration of the second time line. Figure 6.1 demonstrates the process.

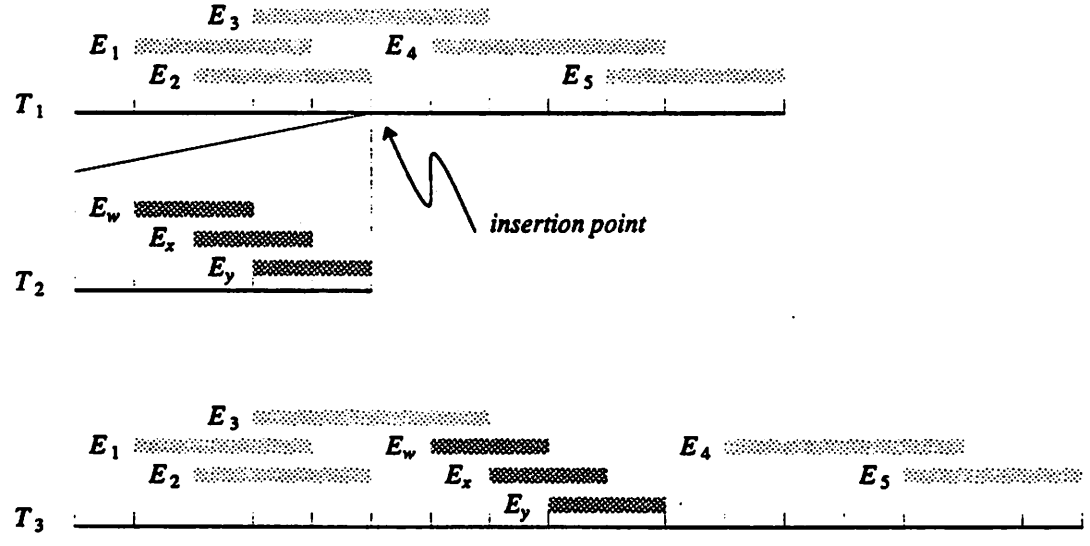


Figure 6.1. The SPLICE-IN Operation

In this example, events on time line T_2 are spliced into time line T_1. The result is called T_3.

This is not the only way in which a splice operation could be defined. For example, we do not consider in this operation any change in the durations of events. Therefore, events which overlap before the splice may not overlap afterward (as for events E_3 and E_4 in figure 6.1).

The process of splicing time lines models real world modifications in schedules. For example, inserting a set of speakers in a conference agenda pushes all future speakers forward in time by a fixed amount. In the musical example, inserting some musical material into a composition causes all future musical events to occur at a later time. Notice that the temporal relationship among these future events remains unchanged, and that events which are simultaneous prior to the operation remain so after the operation.

6.2.2. The Overlay Operation

The second form of merging time lines employs the "overlaying" operation. In this form of update, we identify the time origin of the second time line with a point on the first time line (the *overlay-point*). We then place each event of the second time line onto the first time line by determining the offset of the event start times with respect to their new time origin. All events in the first time line remain unmodified. Figure 6.2 demonstrates the overlaying of one time line onto another. T_2 is overlaid onto T_1, and the result is called T_3.

Overlaying two time lines is intended to model the process of running two event streams in parallel. The time line, in a sense, branches at the overlay-point into two parallel streams of events. We do not however, maintain information regarding this separation, and the overlaid events become indistinguishable from those on the original time line.

Corresponding to the above forms of insertion into the time line, there are two types of deletion in a time line. The first form "splices out" a section of the time line, and the other form eliminates events from an intact time line.

6.2.3. The Splice-out Operation

The process of taking out a section of the time line and splicing together the remaining ends is the inverse of the "splice-in" operation described earlier. We delimit a portion of the time line to be

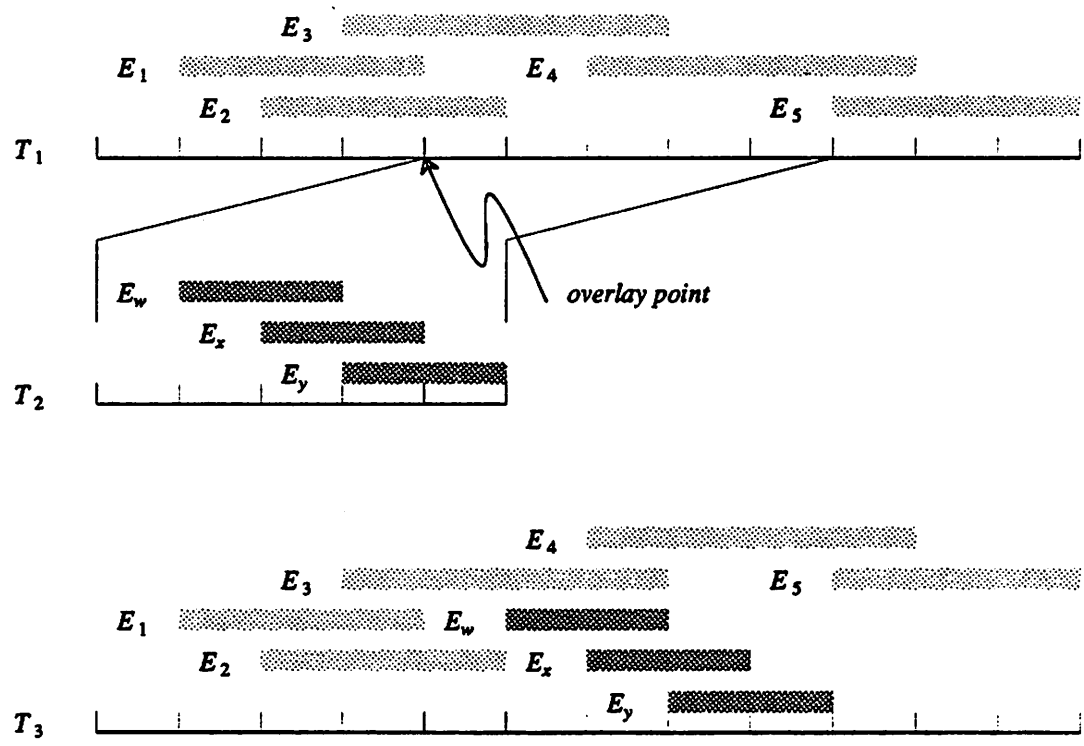


Figure 6.2. The OVERLAY Operation

removed. All events initiated during that time are eliminated. All events initiated after that time are slid earlier in time (i.e. moved up in the schedule). Figure 6.3 shows an example of the splice-out operation.

6.2.4. The Remove Operation

The second form of deletion merely eliminates a set of events from a time line. Unlike the splicing operations, the removal of a set of events does not effect the placement of other events. Figure 6.4 demonstrates the remove-event operation.

In order to obtain the set of events used by an operation such as the *remove* operation, a *retrieve* function must of course be provided.

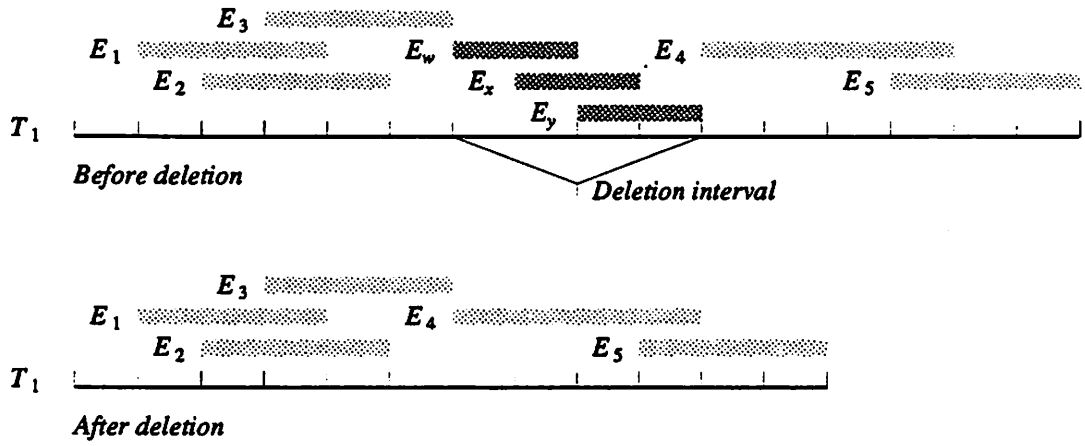


Figure 6.3. The SPLICE-OUT Operation

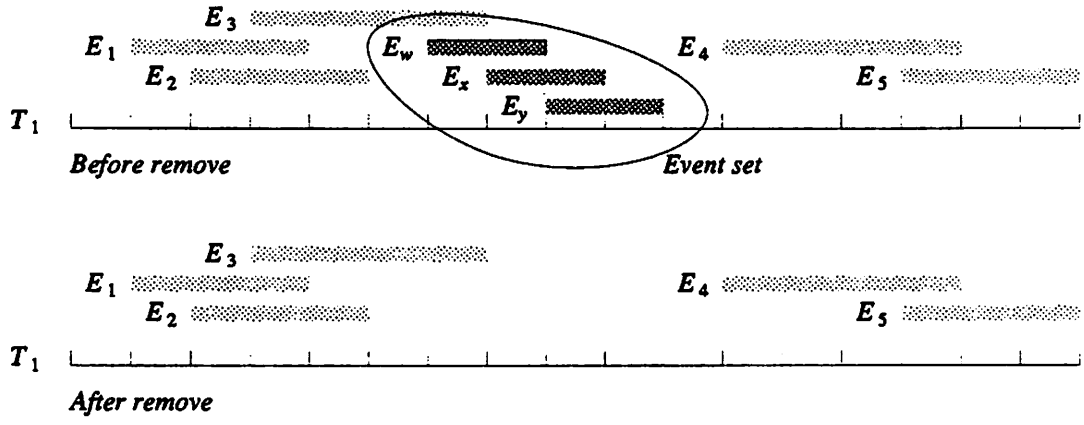


Figure 6.4. The REMOVE Operation

6.2.5. The Interval-retrieve Operation

For completeness, a retrieval operation is defined over time lines. Given a time line and an interval (a start_time and a duration), the "interval-retrieve" operation returns the set of all events on the time line which are initiated during the given interval.

This provides a reasonably complete set of operations to be applied to time lines. The following sections will discuss how time lines may be implemented using A-trees in order to efficiently support these operations.

6.3. Using A-trees to Index Time lines

The definition for events mentioned in the previous section, specifying a native "start-time" attribute, is not a good one for the operations presented. In order to demonstrate its deficiencies, consider the operation of splicing one time line (presumably a small time fragment) into another (presumably a large one). This can be accomplished by the query language program shown in figure 6.5. In general, this set of queries needs to modify nearly every tuple in both time lines. Every event on T_1 after the insertion point is modified in step 2, and every event on T_2 is modified in step 3. We would

---

```
/* T1 and T2 are time lines */
/* insertion_point is a point in T1 time */

/* Step 1: Get bounding length of T2 */
replace TIMELINE (Duration =
     max(EVENT.start_time + EVENT.duration
          where EVENT under T2)
     where TIMELINE is T2

/* Step 2: Slide tail of time line T1 forward in time */
replace EVENT
     (start_time = EVENT.start_time + TIMELINE.duration)
     where TIMELINE is T2
     and EVENT under T1
     and EVENT.start_time > insertion_point

/* Step 3: All the events on T2 get moved to T1 */
/* starting at insertion point */
replace EVENT under T1
     (start_time = insertion_point + EVENT.start_time)
     where EVENT under T2
```

Figure 6.5. Naive Insertion of an Event

---

like to be able to perform operations such as these, which are conceptually local operations (in this case, local to the "insertion point" within T_1) without making these more global modifications to the database (in this case, modifying as much as all of T_1 after the insertion point).

We solve this dilemma with an A-tree index. Given a very large number of tuples in the EVENT relation, we may establish an index which provides rapid access for the splice and retrieval operations.

Under the splicing operations, we notice the following invariants:

- Simultaneous events remain simultaneous.
- The distance between event start times is unchanged for pairs of events which do not span the splicing point.
- For all pairs which span the splicing point, the distance is changed only by a constant.

It can be seen from the above invariants that a good quantity on which to index is the distance (in time) between consecutive events.

Initially, we eliminate all references to start time with respect to the time line origin, and replace them with references to time with respect to the start time of the preceding event. We will call this new attribute the *delay* of an event. In doing this, we have replaced a set of previously independent event placements with a set of interdependent descriptions. The links now provide the ordering information among the events. Figure 6.6 shows a time line and its associated ordered structure. Each entry of the list contains its delay from the previous entry. The start time of an event is now merely the sum of its delay and all preceding delays. The modified definitions are:

define entity TIMELINE (duration = **integer**)
define entity EVENT (delay = **integer**)

define ordering (EVENT) **under** TIMELINE

define inheritance EVENT (start_time = ordered_sum(EVENT.delay))

modify EVENT **to A-tree**

The splicing process now becomes very simple. We need merely insert one set of events between two events in another, making only local changes (including adjustment of the index). The process of inserting delays onto a point on the list implicitly slides the future events forward on the

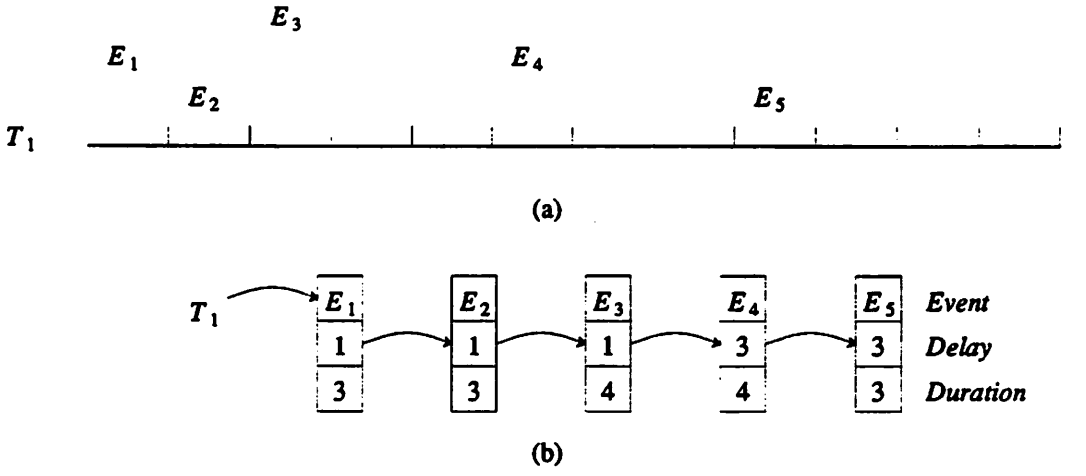


Figure 6.6. Ordered Structure for Time Line

time line.

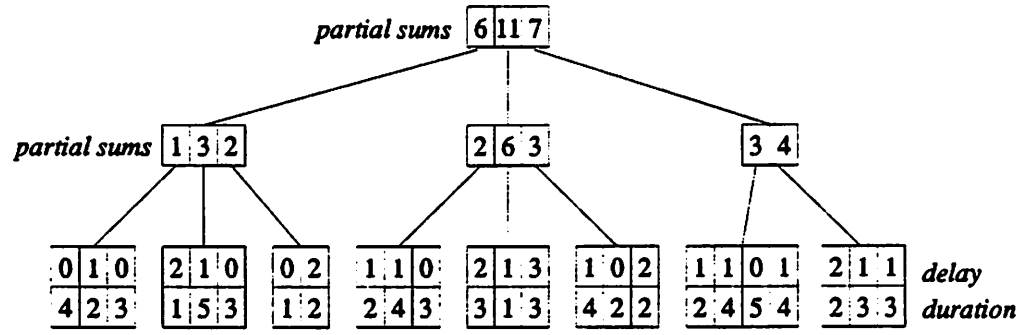
The **modify** statement builds an A-tree index which allows us to efficiently calculate the start_time attribute. We do this by building the tree on top of the event list. The start_time is maintained as an ordered aggregate that sums over the delay attribute. Figure 6.7 shows such a tree over a set of events on a time line. The internal records of this tree contain the summary information for the "ordered_sum" aggregate. For a given internal record, this summary information is the sum of the delay values rooted at that record. At the leaf level, the delays and durations are stored (because this is a primary index, the data records themselves are stored, rather than pointers to the data records). For reference, the diagram of events represented by this particular tree is also given.

6.4. Implementing Time Line Operations

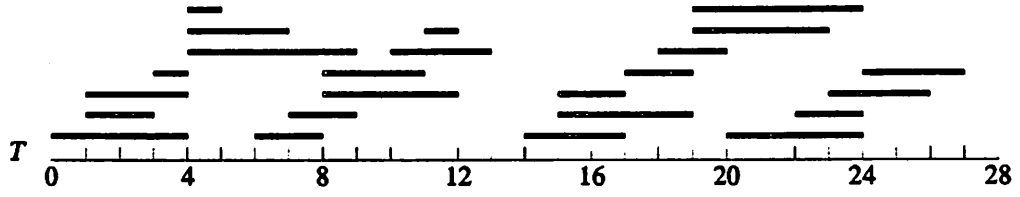
What follows is a detailed description of the query language programs necessary to implement time lines and events, and efficiently support the splice and overlay operations.

6.4.1. Inserting Events

Four pieces of information are needed to install an event: the name of the event, e, the name of the time line on which it is to be installed, t, the start time of the event, s, and the duration of the event, d. Given these constants, the commands shown in figure 6.8 insert the event. The first step



(a)



(b)

Figure 6.7. An A-tree and The Events Which It Indexes

```
/* find the predecessor of e at s */
retrieve into PREDEVENT (event = EVENT, olddelay = EVENT.delay)
     where EVENT.start_time = last(EVENT.start_time
          where EVENT.start_time < s))
     and EVENT under t

/* update its delay */
replace EVENT (delay = s - EVENT.start_time)
     where EVENT is PREDEVENT.event

/* install the new event */
range of e1 is EVENT
append to EVENT after e1 under t
     (delay = e1.start_time + PREDEVENT.old_delay - s,
     duration = d)
     where e1 is PREDEVENT.event
```

Figure 6.8. Inserting an Event

determines the predecessor of e, and the second step alters its delay to reflect the fact that a new event is being inserted. The third step actually inserts the new event e with the correct delay value.

6.4.2. Implementing the Overlay Operation

To overlay the events on a time line T_2 onto another time line T_1, we want to preserve start times of all the events. The simplest way to do this is to select the small number of events in each time line that are local to the insertion, and materialize their start times in a temporary relation. This allows us to perform insertions without automatically updating the virtual start time attributes. This temporary relation will be called OVERLAY. Here is the fragment of the query language that implements the overlay operation.

```
define entity OVERLAY (event = EVENT, start_time = integer)
define ordering (OVERLAY)
define inheritance OVERLAY (ordinate = ordered_count(OVERLAY))

range of e1, e2 is EVENT
range of o1, o2 is OVERLAY

/* slide T2 forward by Overlay point */
replace EVENT (delay = overlay_point + EVENT.delay)
    where EVENT is first(EVENT under T2)


/* get into a temporary relation the relevant events of T1,
materializing their start_time attribute values in the process */

retrieve into OVERLAY (event= e1, start_time = e1.start_time)
    where e1 under T1
    and e2 under T2
    and e1.start_time >= first(e2 under T2).start_time
    and e1.start_time <= last(e2 under T2).start_time

/* add T1 boundary events to OVERLAY */
append into OVERLAY (event=e1, start_time=e1.start_time)
    where e1 under T1
    and e1.start_time = last(e1 under T1
            where e1.start_time < first(OVERLAY).start_time).start_time
```

```
append into OVERLAY
    (event = e1, start_time = e1.start_time)
    where e1 under T1
    and e1.start_time = first(EVENT.start_time under T1
        where e1.start_time > last(OVERLAY).start_time).start_time
    and o1 is last(o1 where o1.start_time < e1.start_time)

/* add T2 events to OVERLAY */
append into OVERLAY
    (event = e2, start_time = e2.start_time)
    where e2 under T2


/* having done all the appends, reorganize OVERLAY */
reorder OVERLAY by start_time

/* Update EVENT to reflect events in OVERLAY */
replace EVENT under T1
    (delay = o2.start_time - o1.start_time)
    where EVENT is o2.event
    and o1.ordinate = o2.ordinate - 1

destroy OVERLAY
```

This program first creates the temporary relation, OVERLAY as an ordered relation. We will fill the OVERLAY relation with events from the EVENT relation. Notice that, for each event, start time is a virtual attribute provided by an ordered aggregate over the hierarchically ordered EVENT relation. In the OVERLAY relation, on the other hand, start_time is a physical (materialized) attribute.

A **replace** statement then slides all the start times on T_2 forward by the overlay point. This requires only a single update to the first *delay* value.

The next four statements fill the OVERLAY relation. First, we collect all the relevant events in T_1, those that overlap the time space of T_2 (as modified). The next two statements each add a single additional event to the OVERLAY relation at the boundaries of the T_2 time space. We then add all of the events in T_2 to the OVERLAY relation. Because we did not specify where in the OVERLAY ordering these appended records should be placed, their ordering is at this point unknown. We explicitly reorder the OVERLAY records, by sorting them on their start times.

We then go back to the EVENT relation, and update all the events that are involved in the OVERLAY time space. The delays are easily calculated, since the OVERLAY relation orders the start times of successive events.

6.4.3. Implementing the Splice Operation

In splicing one time line into another, the relative distance between events (that is, their *delay*) remains constant. We can use the A-tree over the EVENT relation to automatically maintain the appropriate values for each event's start time.

To splice one time line into another, the following QUEL fragment is executed. It splices the events in T_2 into time line T_1 at the given *insertion_point.*

```
range of e1,e2 is EVENT

replace e2 after e1 under T1
    where e1 under T1
    and e2 under T2
    and e1 is last(EVENT under T1
        where EVENT.start_time < insertion_point)
```

This splice operation can be done in a single query. We merely transfer events from T_2 to T_1, inserting them after the insertion point.

6.5. Using Inheritance to Define Time Maps

An important property of musical events is that they are composed using one time frame (score time) and performed using a slightly different one (performance time). In an orchestral concert, the conductor is typically responsible for coordinating the translation of score time into performance time. This notion of translation between time frames is formalized in [Jaf85] through the concept of time and tempo maps.

This discussion concludes with a demonstration that, by integrating our inheritance specifications with the ordered aggregate attributes provided by the A-tree index, we can implement tempo and time maps as presented in [Jaf85].

We begin by reviewing some definitions. "Score time" is the temporal point at which events are scheduled to begin in an abstract time space (for example, the fourth beat of the sixtieth measure of a composition). "Performance time" is the actual time at which the event is performed (for example, forty seconds after the start of the composition). In this system, the tempo (i.e. rate of score time passage with respect to real time) of a set of events may be specified as an arbitrary function of score time (see figure 6.9). The upper graph in this figure shows tempo as a function of time. This constitutes a

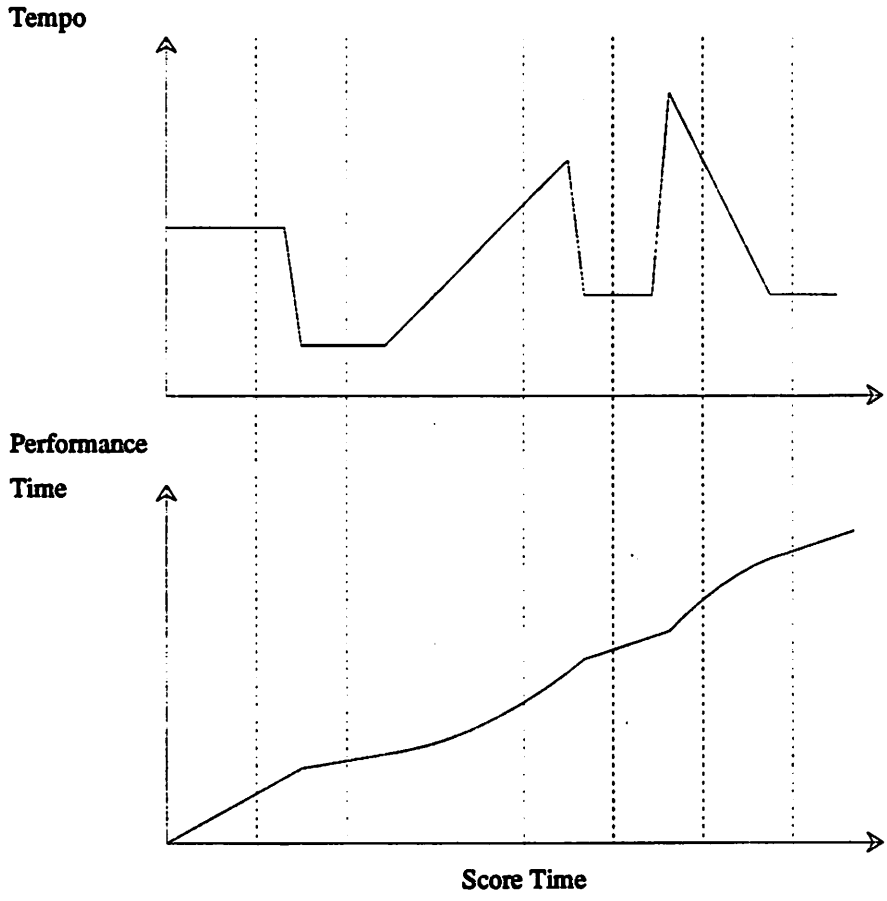


Figure 6.9. Tempo and Time Maps

tempo map. In a CMN score, annotations such as *presto* (fast) or *lento* (slow) indicate absolute tempo. The flat portions of the graph represent those tempo settings. Other annotations, such as *accelerando* (accelerate) and *rallentando* (slow down) signify a tempo that changes over time. The sloped portions of the graph show this type of tempo indication.

The lower graph results from integrating the tempo function over score time. Since the tempo function maps the rate of change of performance time at each point in score time, the integration in the lower graph represents the point in performance time corresponding to a particular point in score time. This is a *time map*.

A tempo map therefore represents a flexible transformation from one time frame into another. The utility of this mapping lies in the fact that it preserves simultaneity. Events that are simultaneous

in the original time frame remain simultaneous in the resultant time frame (the actual point in the resultant time frame at which the events occur is determined by the particular tempo map).

A tempo map is represented in the database by a relationship between points in score time (the time units in which the events are placed on time lines), and the tempo at those points. If the tempo is specified at regular intervals, the following relation suffices:

```
define entity TEMPOMAP (tempo = integer)
define ordering (TEMPOMAP)
```

The time map can be derived from this tempo map by integration, in other words, by summing over the tempo values. The A-tree structure to accomplish this is:

```
define inheritance MAP
    score_time = ordered_count(MAP),
    performance_time = ordered_sum(MAP.tempo))
```

Now, given a events and tempo maps, we can define the performance time of these events using the inheritance definition shown in figure 6.10. This definition adds two attributes to the EVENT entity: "performance_start_time," and "performance_duration." They are calculated dynamically based on the values of the start_time in the EVENT relation, and the time mapping indicated by the score_time and the performance_time in the MAP entity.

Using this mechanism, the MAP may be modified at will by the user, and the start times of all mapped time attributes will be automatically maintained by the system. The only native attributes required by the calculation are the delay and duration attributes for events, and the tempo attribute at each discrete point in time. From these, start_time, performance_start_time, and performance_duration are all determined.

```
range of M1, M2 is MAP

define inheritance EVENT
    (performance_start_time = M1.performance_time,
    performance_duration = M2.performance_time - M1.performance_time)
    where EVENT.start_time = M1.score_time
    and EVENT.start_time + EVENT.duration = M2.score_time
```

Figure 6.10. Inheriting Performance Time

6.6. Summary

Temporal information in the database has two faces: the temporal aspect of insertion and update in the database (operations which take place at points in time), and the temporal semantic content of the data itself. Although much research in modeling temporal information in the database has applied to the former aspect, we are concerned here with the latter.

We have presented a detailed application of A-trees to the management of temporal information. In this model, the unit of temporal information is the *event*. Events are hierarchically ordered under particular *time lines*.

Several operations are defined on these time lines:

- Splicing one time line into another,
- Overlaying one time line onto another,
- Deleting a section of a time line,
- Removing a set of events from a time line,
- Retrieving a set of events on a time line.

After defining the A-tree used to index the event relation, the QUEL program fragments for splicing and overlaying time lines are given.

Music uses many different kinds of time, such as score time, score time, and performance time. The mapping between these time spaces may be specified using *time maps* [Jaf85]. These time maps themselves may be defined as ordered relations. We have shown how a relational view definition built on top of the ordered event relation can provide flexible mapping from one time space to another.

CHAPTER 7

Conclusion

7.1. Summary of Research

This research has focused on two major areas in the development of a data manager for a musical database. The first is a data model for musical information, and the other is a strategy for effectively implementing this model. What follows is a summary of our research on these issues.

7.1.1. Data Modeling

Before a reasonable musical information manager can be constructed, a model of musical information itself is required. The tools for building this model, and many aspects of the model itself, have been the focus of this research.

Fundamental aspects of musical information, either as CMN scores or event time lines, incorporate the concepts of *order* and *hierarchy*. The interaction between these two concepts provides a basis for *Ordered Aggregation Hierarchies*, which serve as our primary tool for modeling musical information.

These hierarchies provide a framework around which we organize *attribute inheritance*. Entities inherit attribute values by performing various computations on the attributes of their parents, siblings, and children in their hierarchies. These related entities in turn may inherit attributes from their immediate relatives. In this way, a very rich structure is provided for propagating attribute values through the hierarchy.

7.1.2. Implementation Strategies

Upon developing this data model, it became clear that access methods currently available in database systems are insufficiently powerful to support it. Using the entity-relationship model as a starting point, a number of extensions to a relational database system were developed to support the efficient use of hierarchical ordering.

Underlying these extensions is a new relation type, the *ordered relation*, and a new access method, the A-tree. A-trees allow for the efficient calculation of a large class of aggregate functions over ordered relations. These functions are named *ordered aggregates*. The client interface developed for A-trees permits a high degree of control by the client over the specific use of the access path, allowing the implementation of a wide variety of ordered aggregates. We presented examples of line numbers, running balances, and exponential averages to indicate a variety of uses for ordered aggregation.

In addition to supporting the notions of hierarchy and ordering in the data, A-trees solve, in part, the problem of efficiently implementing complex attribute inheritance. This is possible when the entities participating in the inheritance function are the aggregation of siblings of a given entity.

For more general types of inheritance, there is no alternative but to provide some means of storing functional specifications (i.e. procedural data) in the database. We have modified a relational view mechanism to support these specifications. When a query references an inherited attribute, the query is modified to compute the value of the attribute based on its inheritance specification. This may be efficiently implemented by precomputing such values and caching the results in the database.

7.2. Further Research

In an obvious sense, this dissertation is incomplete. It intends to develop a data manager for musical information, yet only addresses a subset of the issues presented by such a system. In order to demonstrate the viability of such a tool, the following tasks need be done:

- Build the model extensions and access methods into an actual relational system.

It is possible to use a standard relational database to store musical information; the extensions to the relational model suggested by this thesis affect the *naturalness* of the representation, and so affect the *efficiency* of utilizing the information. These extensions need to be incorporated into an actual relational system to determine there effectiveness.

Although simulations and analysis indicate that particular performance improvements may be realized by the access methods developed by this research, they have yet to be used under real conditions. A "road test" of these techniques is required.

- Develop front end tools for the data manager.

Such tools include a score editor or event sequencing tool, or perhaps analysis programs. Any existing program that operates on musical information could in theory be modified to use the database back end. This might prove to be a simple path to actual usage of the data manager.

- Assess the performance of the data manager.

Given a user program (such as an analysis program) that has been modified to read its musical information input out from the data manager, it is a simple exercise to compare its performance to the same program operating on its own data. One would expect a performance penalty in using a general purpose data manager (in place of application-specific code). It remains to be seen whether or not this penalty is acceptable.

APPENDIX A

A Music Font

This appendix contains the set of icons and linears (as described in Chapter 2) used in the prototype music database developed in the course of our research. Most of these objects are referenced explicitly by the schema given in appendix C.

The various objects are presented in their outline form, to make their construction more clearly visible. They are represented as Postscript procedures [Ado85],, and may thus be scaled to any size, and rendered on a variety of graphics devices.

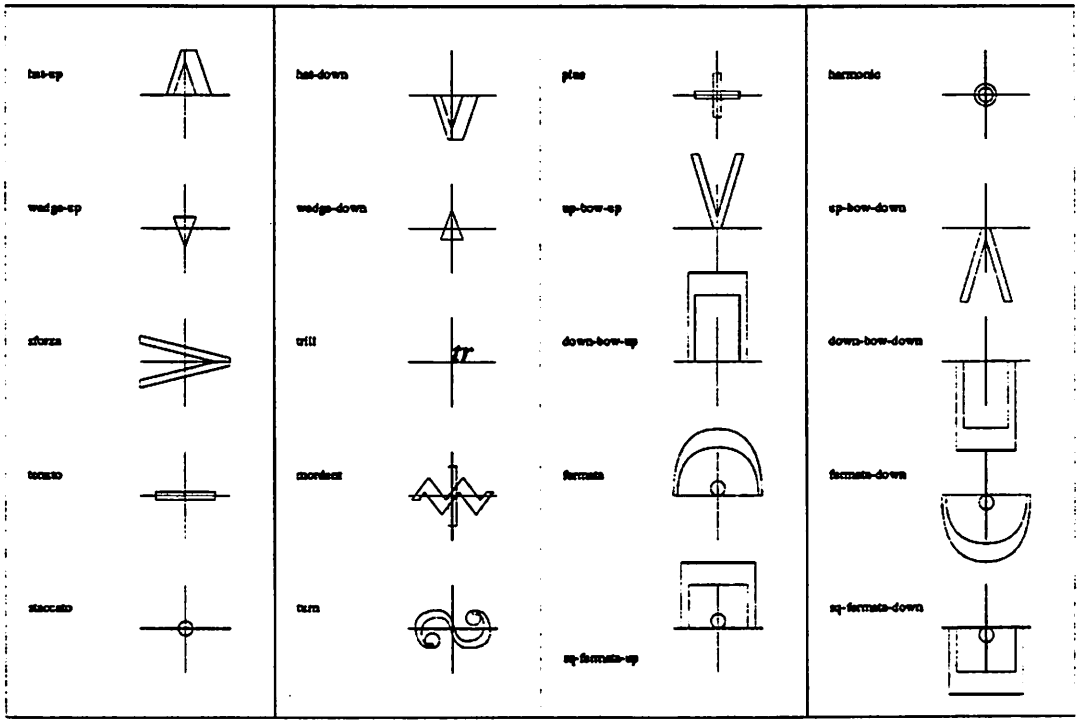


Figure A.1. Accents

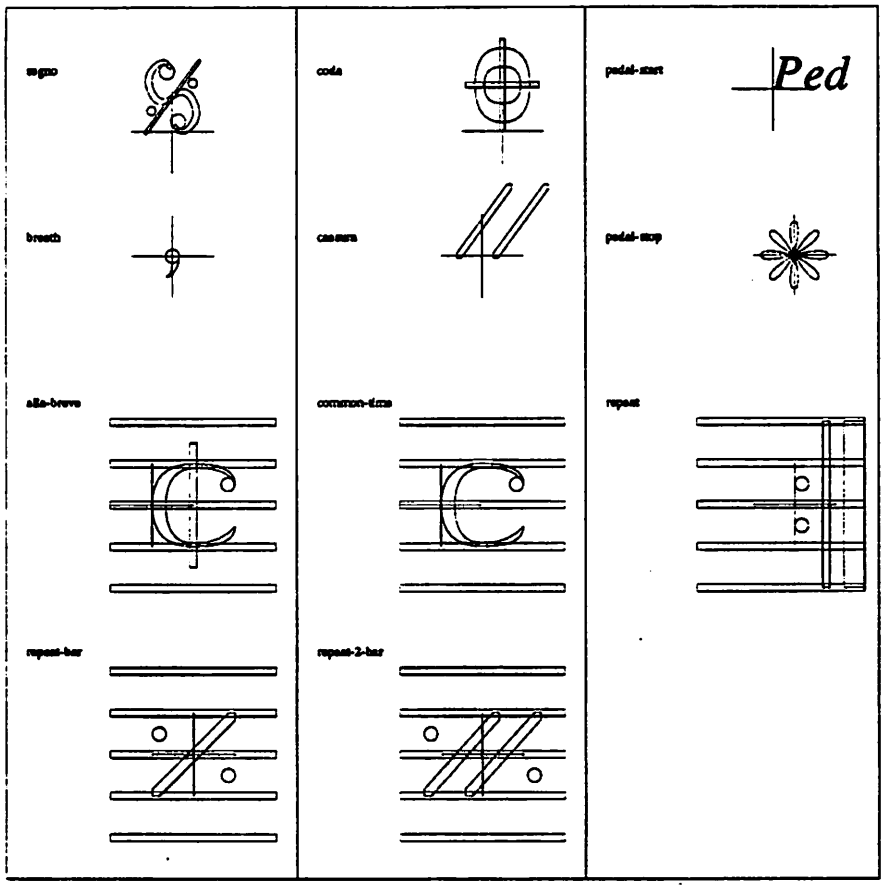


Figure A.2. Annotations

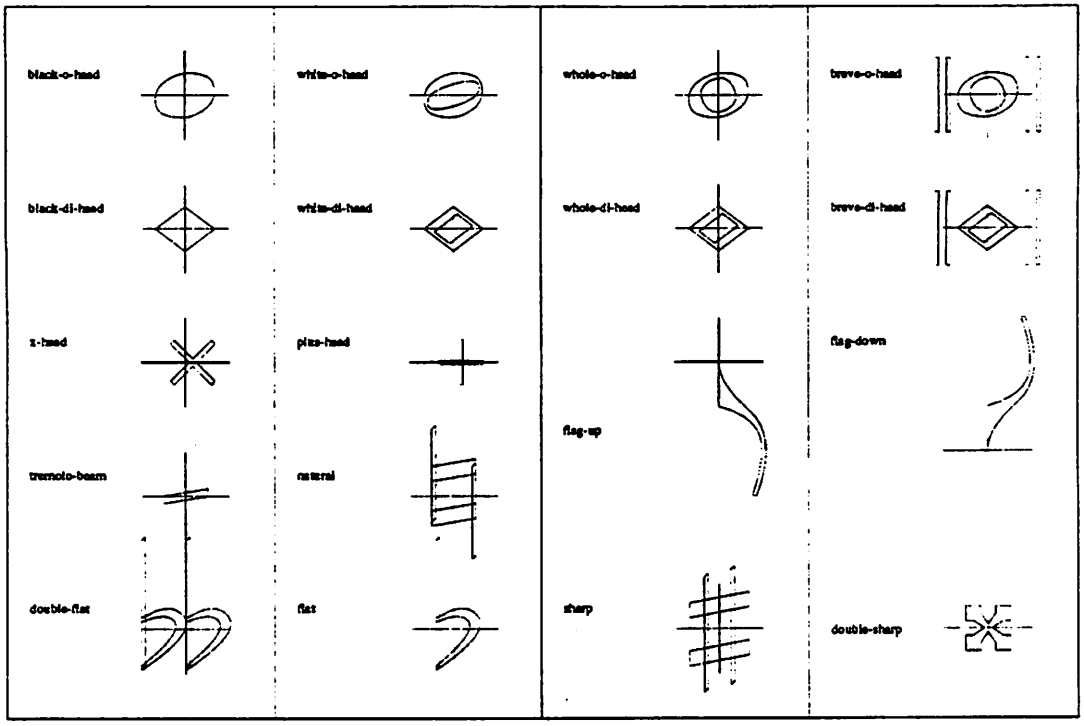


Figure A.3. Chord and Note Parts

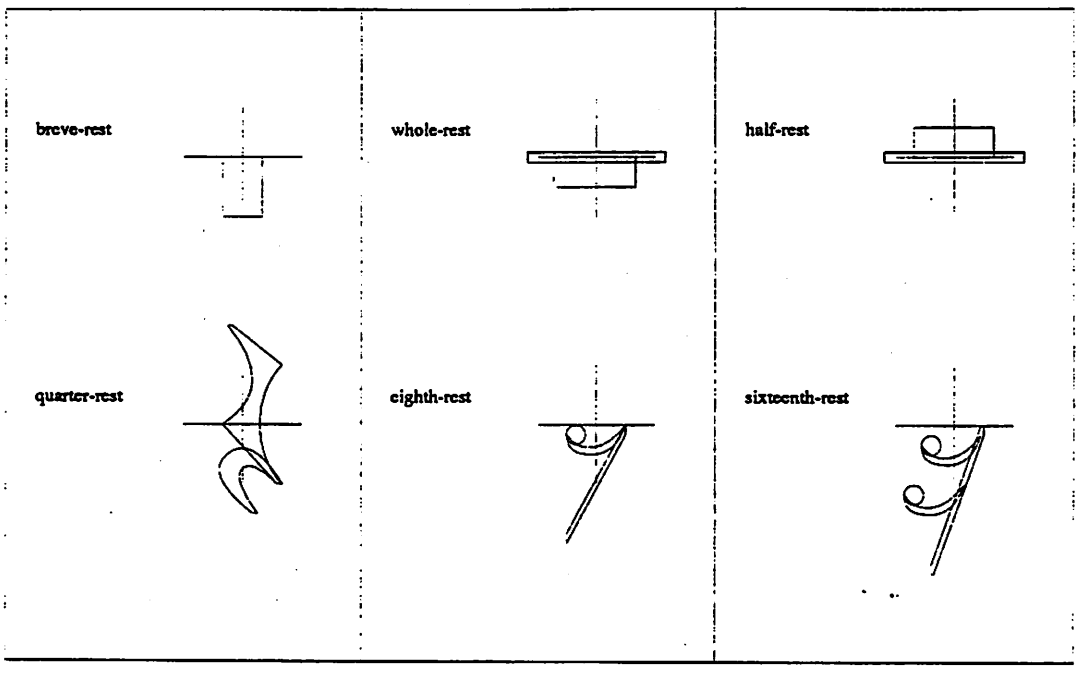


Figure A.4. Rests

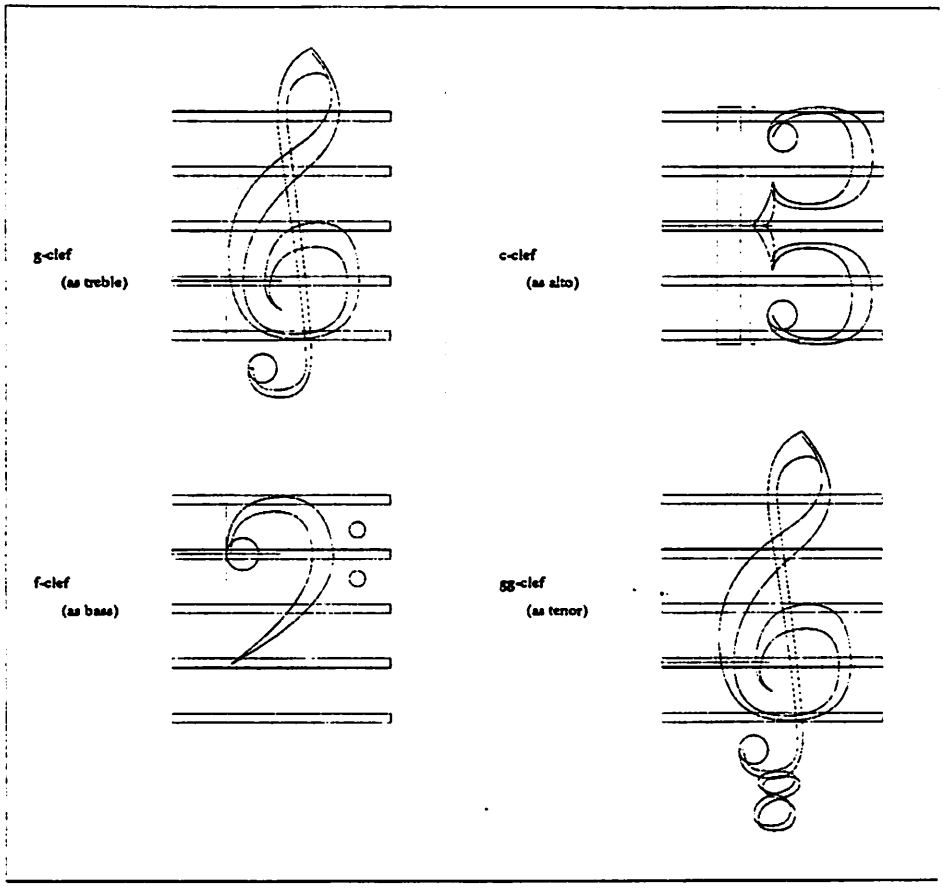


Figure A.5. Clefs

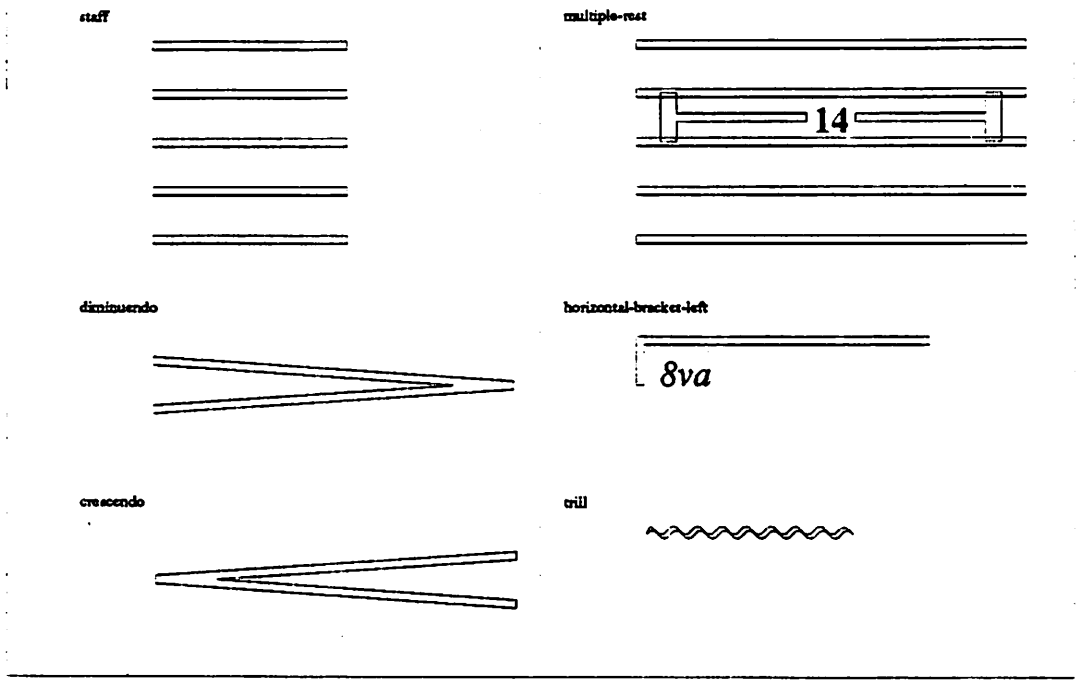


Figure A.6. Horizontal Linears

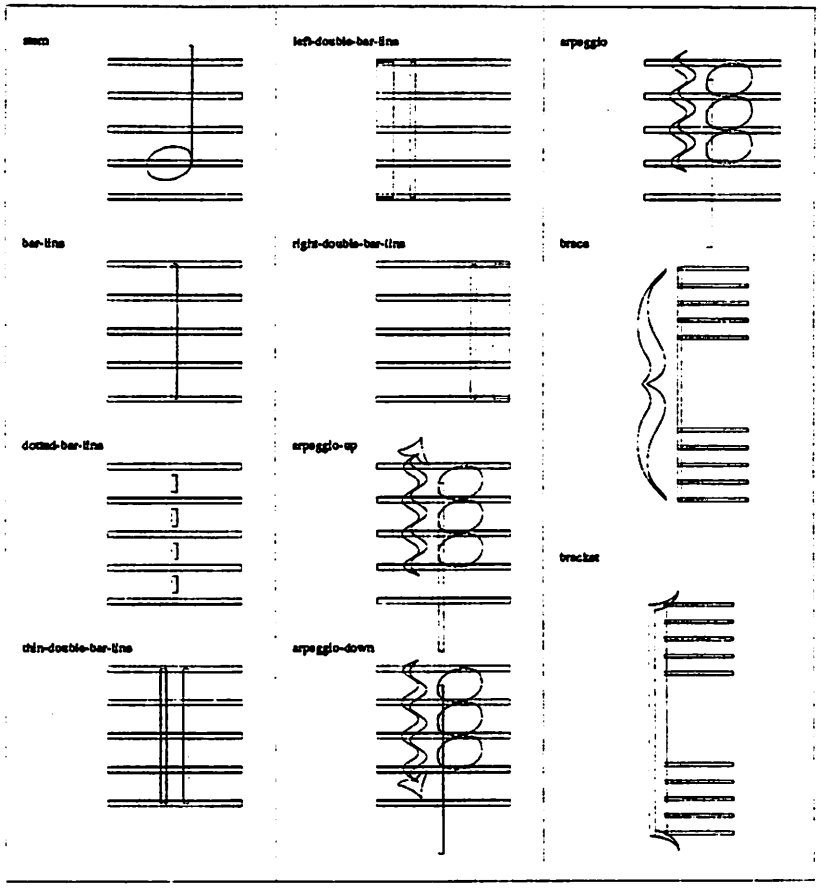


Figure A.7. Vertical Linears

APPENDIX B

An Example of Update to Inherited Attributes

An example of update to inherited attributes was presented in chapter 3. This appendix contains the actual program fragment to perform the update.

In this example, we wish to modify the start time of a chord c_1 to be synchronous with the start time of chord c_2. The program is written using pseudo-code (primarily to represent control structures) intermixed with QUEL.

Here is an overview of the program:

Find chords before and after c_1 in its voice
Move c_1 to new group, and fix instance graph
Get a group for c_1 to go into
Find chords before and after c_1 in sync
Move c_1 into new sync, and fix instance graph
Recalculate durations of s_{old} and s_{new}
Recalculate start times of syncs, and pass them downward
Recalculate start times of groups in movement

Here is the program fragment to set the start time of chord c_1 equal to that of chord c_2:

```
/* find chords before and after c1 in voice */
find c_before with maximum start_time
     where chord.start_time < c2.start_time
     and chord.voice_parent = c1.voice_parent

find c_after with minimum start_time
     where chord.start_time > c2.start_time
     and chord.voice_parent = c1.voice_parent

/* move c1 to new group, and fix instance graph */
g_old = c1.group_parent
/* get a group for c1 to go into */
if c_before.group_parent != c_after.group_parent then
   make a new group g_new
   g_new.start_time = c2.start_time
   g_new.duration = c1.duration
   g_new.voice_parent = c1.voice_parent
   g_new.voice_ordinate = c_before.voice_ordinate + 1
   replace group (voice_ordinate = group.voice_ordinate + 1)
      where group.voice_parent = g_new.voice_parent
      and group.voice_ordinate > g_new.voice_ordinate
   c1.group_parent = g_new
   c1.group_ordinate = 1
else
```

```
   g_new ← c_sub_before .group_parent
   c_1.group_parent = g_new
   c_1.group_ordinate = c_before.group_ordinate + 1
   replace chord (group_ordinate = chord.group_ordinate + 1)
      where chord.group_parent = c_1.group_parent
      and chord.group_ordinate > c_1.group_ordinate

/* find chords before and after c_1 in sync */
find c_before with maximum sync_ordinate
      where chord.sync_ordinate is < c_2.sync_ordinate
      and chord.sync_parent = c_2.sync_parent

find c_after with minimum sync_ordinate
      where chord.sync_ordinate is > c_2.sync_ordinate
      and chord.sync_parent = c_2.sync_parent

/* move c_1 into new sync, and fix instance graph */
s_old = c_1.sync_parent
s_new = c_2.sync_parent
c_1.sync_parent = s_new
c_1.sync_ordinate = c_before.sync_ordinate + 1

replace chord (sync_ordinate = chord.sync_ordinate + 1)
   where chord.sync_parent = c_1.sync_parent
   and chord.sync_ordinate > c_1.sync_ordinate

/* recalculate durations of s_old and s_new */
replace sync (duration = min(chord.duration where
      chord.sync_parent = s_old))
   where sync.uid is s_old

replace sync (duration = min(chord.duration where
      chord.sync_parent = s_new))
   where sync.uid is s_new

/* recalculate start times of syncs, and pass them downward */
for every sync s
   where s .movement_parent = c_1.movement_parent:

   replace sync (start_time = sum(sync.duration
         where sync.movement_parent = s .movement_parent
         and sync.movement_ordinate < s .movement_ordinate))
      where sync.uid = s

   replace chord (start_time = s .start_time)
      where chord.sync_parent = s

   replace note (start_time = s .start_time)
      where note.sync_parent = s

/* recalculate start times of groups in movement */
for every group g
   where g .movement_parent = c_1.movement_parent:

   replace group (start_time = min(chord.start_time
         where chord.group_parent = g ))
      where group.uid = g
```

APPENDIX C

Musical Database Schema

The following database schema was developed to represent CMN scores and MIDI information as the basis for a musical information database. It is based loosely on an entity-relationship model.

The following presentation is broken into three parts:

- The data types used by the schema,
- The relations, each with its attributes (both native and inherited),
- The indices providing appropriate access paths, as well as ordered aggregate attribute values, for the relations.

C.1. Data Types

Figure C.1 summarizes the data types used by the schema.

Str and *Num* should be self-explanatory. A *Date* is a six byte strings that map to a day, month and year. Attributes of type *Ref*, reference fields, are keys, typically entity uid's, that are imported into a relation. They allow a record to "point" to a record in another entity. *Text* and *Binary* data types are variable length byte strings. The only distinction between these two types lies in their use, rather than their representation. Text strings are typically used to store human readable information, such as program fragments. Binary data is typically machine readable, as are, for example, MIDI command strings.

The remaining types have more complex underlying representations, but they have fixed sizes and interpretations. A *Time* element is either a duration, or, interpreted relative to a time line origin, a point on some time line. A *Tspan* is the portion of a time line delimited by a pair of time points.

There are three graphical types. A *Dist* is a distance in graphical space. A point is a pair of (x,y) distances from the graphical origin. A *Bbox* is the bounding box defined by a pair of points indicating two opposite corners of the rectangle (typically the one closest to the graphical origin, and the one furthest).

Type	Length	Description
Str	*	Fixed length character strings
Num	*	Integer numbers
Date	6	Calendar dates
Ref	†	Reference fields
Text	§	Arbitrary length text (ASCII)
Binary	§	Arbitrary length byte strings
Time	4	A temporal duration or point in time
Tspan	8	A time span
Dist	4	A graphical distance
Point	8	A point in graphical space
Bbox	16	A bounding box in graphical space
Degree	1	A staff degree
Pitch	2	A musical pitch (MIDI)

* The length of these attributes is fixed in the schema
† The length of a reference attribute is the length of the imported key (see text).
§ The length of these attributes varies dynamically as the objects are created and modified

Figure C.1. Summary of Data Types

A *Degree* refers to vertical positions on the staff. The bottom line of a staff is degree zero, and each line and space above the staff is numbered consecutively. Positions below the staff are represented by negative numbers. Finally, *Pitch* is an enumerated type that may be represented in two bytes.

C.2. Relations and Attributes

The following (long) table consists of all relations and their attributes. For every attribute, its type and byte length are given. For reference attributes, the relation and key name to be imported are given. For inherited attributes the inheritance function is given. Some of these inheritance functions are single queries, others are arbitrary database procedures. In the latter case, the name of the procedure is given. Key fields are marked with an asterisk.

In general every entity that exists on a page has a bounding box. It may also have a normative point indicating its logical position on the page. Every movement has its own timeline. Every entity

that exists on a time line has a time span and a normative time point associated with it. Associated with each ordered aggregation are three attributes: A count attribute in the parent entity, and a reference pointer to the parent and ordinate position in the child entity. Of course, not every reference pointer exists because of a ordered aggregation construct.

Ordering the entities of the schema in a listing such as this one is problematic, since most entities belong to several different groups, and thus they do not admit to a linear ordering. In general, the more abstract entities come first, followed by lower level objects below.

Score				
	*uid	Num	4	
	movement_cnt	Num	4	
	page_cnt	Num	4	
Composer				
	*uid	Num	4	
	name	Str	50	
	born	Date	6	
	died	Date	6	
	country	Str	20	
Biblio				
	score	Ref	4	Score/uid
	composer	Ref	4	Composer/uid
	title	Str	40	
	subtitle	Str	40	
	compositiondate	Date	6	
	publicationdate	Date	6	
	refid	Str	40	
GraphDef				
	*uid	Num	4	
	name	Str	40	
	postscript	Text	40	
Page				
	*uid	Num	4	
	score_par	Ref	4	Score/uid
	score_ord	Num	4	
	system_cnt	Num	4	
	page_number	Num	4	
	bbox	Bbox	16	
	tspan	Tspan	8	

Movement				
	*uid	Num	4	
	score_par	Ref	4	Score/uid
	score_ord	Num	4	
	first_page	Ref	4	Page/uid
	last_page	Ref	4	Page/uid
	measure_cnt	Num	4	
	instrum_cnt	Num	4	
	system_cnt	Num	4	
	tspan	Tspan	8	
Annot				
	*uid	Num	4	
	label	Str	100	
	Page	Ref	4	Page/uid
	location	Point	4	
	size	Num	2	
	font	Ref	4	Font/uid
	position	Ref	4	Position/uid
Font				
	*uid	Num	4	
	name	Str	40	
Position				
	*uid	Num	4	
	name	Str	20	
	hfactor	Num	2	
	vfactor	Num	2	
AnScore				
	score_parent	Ref	4	Score/uid
	score_ordinate	Num	4	
	annot	Ref	4	Annot/uid
InstDef				
	*uid	Num	4	
	name	Str	40	
	type	Str	20	
	low	Num	4	
	high	Num	4	
	description	Text	--	
AnMove				
	movement_par	Ref	4	Movement/uid
	movement_ord	Num	4	
	annot	Ref	4	Annot/uid
Measure				
	*uid	Num	4	
	movement_par	Ref	4	Movement/uid
	movement_ord	Num	4	
	sync_cnt	Num	4	
	score_par	Num	4	
	tspan	Tspan	8	
	measure_number	Num	4	

AnPage				
	annot	Ref	4	Annot/uid
	pageref	Ref	4	Page/uid
System				
	*uid	Num	4	
	page_par	Ref	4	Page/uid
	page_ord	Num	4	
	movement_par	Num	4	
	movement_ord	Num	4	
	section_cnt	Num	4	
	sync_cnt	Num	4	
	bbox	Bbox	16	
	tspan	Tspan	8	
Sync				
	*uid	Num	4	
	system_par	Ref	4	System/uid
	system_ord	Num	4	
	measure_par	Ref	4	Measure/uid
	measure_ord	Num	4	
	syllable_cnt	Num	4	
	chord_cnt	Num	4	
	rest_cnt	Num	4	
	bbox	Bbox	16	
	location	Point	8	
	time	Time	4	
Barline				
	sync	Ref	4	Sync/uid
	graphdef	Ref	4	GraphDef/uid
	measure	Ref	4	Measure/uid
Section				
	*uid	Num	4	
	system_par	Ref	4	System/uid
	system_ord	Num	4	
	instrum_cnt	Num	4	
	bbox	Bbox	16	
	tspan	Tspan	8	
Instrum				
	*uid	Num	4	
	instdef	Ref	4	InstDef/uid
	movement_par	Ref	4	Movement/uid
	movement_ord	Num	4	
	part_cnt	Num	4	
	systinst_cnt	Num	4	
	tspan	Tspan	8	
	annot	Ref	4	Annot/uid

Duration				
	*uid	Num	4	
	notehead	Ref	4	GraphDef/uid
	resthead	Ref	4	GraphDef/uid
	w_num	Num	4	
	w_denom	Num	4	
	flags	Num	1	
	dots	Num	1	
	stem_p	Str	1	
SystInst				
	*uid	Num	4	
	instrum_par	Ref	4	Instrum/uid
	instrum_ord	Num	4	
	section_par	Ref	4	Section/uid
	section_ord	Num	4	
	bbox	Bbox	16	
	tspan	Tspan	8	
Staff				
	*uid	Num	4	
	systinst_par	Ref	4	SystInst/uid
	systinst_ord	Num	4	
	text_cnt	Num	4	
	bbox	Bbox	16	
	gr_spaceheight	Num	4	
	tspan	Tspan	8	
Metersig				
	sync	Ref	4	Sync/uid
	staff	Ref	4	Staff/uid
	graphdef	Ref	4	GraphDef/uid
	beats	Num	4	
	per	Ref	4	Duration/uid
AnSystem				
	sync	Ref	4	Sync/uid
	annot	Ref	4	Annot/uid
Part				
	*uid	Num	4	
	instrum_par	Ref	4	Instrum/uid
	instrument_ord	Num	4	
	tspan	Tspan	8	
	annot	Str	4	
KeyDef				
	*uid	Num	4	
	tonality	Str	1	
Keysig				
	staff	Ref	4	Staff/uid
	sync	Ref	4	Sync/uid
	keysig_def	Ref	4	KeyDef/uid
KeyDefg				
	keydef	Ref	4	KeyDef/uid
	accidental	Ref	4	GraphDef/uid
	degree	Degree	1	

Voice				
	*uid	Num	4	
	part_par	Ref	4	Part/uid
	part_ord	Num	4	
	group_cnt	Num	4	
	event_cnt	Num	4	
	tspan	Tspan	8	
Degree				
	*uid	Num	4	
	staff_par	Ref	4	Staff/uid
	staff_ord	Num	4	
	location	Point	4	
	bbox	Bbox	16	
	staffline	Ref	4	GraphDef/uid
	degree	Degree	1	
Group				
	*uid	Num	4	
	voice_par	Ref	4	Voice/uid
	voice_ord	Num	4	
	chord_cnt	Num	4	
	rest_cnt	Num	4	
	tspan	Tspan	8	
Clef				
	graphdef	Ref	4	GraphDef/uid
	degree	Ref	4	Degree/uid
	sync	Ref	4	Sync/uid
Event				
	*uid	Num	4	
	degree	Ref	4	Degree/uid
	voice_par	Ref	4	Voice/uid
	voice_ord	Num	4	
	perf_volume	Num	4	
	perf_pitch	Num	4	
Text				
	*uid	Num	4	
	staff_par	Ref	4	Staff/uid
	staff_ord	Num	4	
AnStaff				
	annot	Ref	4	Annot/uid
	staff	Ref	4	Staff/uid
	sync	Ref	4	Sync/uid
Beam				
	beam_gd	Ref	4	GraphDef/uid
	group_ref	Ref	4	Group/uid

Chord				
	*uid	Num	4	
	sync_par	Ref	4	Sync/uid
	sync_ord	Num	4	
	group_par	Ref	4	Group/uid
	duration	Ref	4	Duration/uid
	tspan	Tspan	8	
	location	Point	8	
	bbox	Bbox	16	
	group_ord	Num	4	
Accent				
	accent_gd	Ref	4	GraphDef/uid
	chord	Ref	4	Chord/uid
Flag				
	flag_gd	Ref	4	GraphDef/uid
	chord	Ref	4	Chord/uid
Note				
	*uid	Num	4	
	event_par	Ref	4	Event/uid
	event_ord	Num	4	
	chord_par	Ref	4	Chord/uid
	event_chord	Num	4	
	location	Point	8	
	tspan	Tspan	8	
Dot				
	dot_gd	Ref	4	GraphDef/uid
	note	Ref	4	Note/uid
Accident				
	note	Ref	4	Note/uid
	accid_gd	Ref	4	GraphDef/uid
Rest				
	*uid	Num	4	
	sync_par	Ref	4	Sync/uid
	sync_ord	Num	4	
	group_par	Ref	4	Group/uid
	group_ord	Num	4	
	degree	Ref	4	Degree/uid
	duration	Ref	4	Duration/uid
	rest_gd	Ref	4	GraphDef/uid
Stem				
	stem_gd	Ref	4	GraphDef/uid
	chord	Ref	4	Chord/uid
	gr_length	Num	4	
DynDef				
	*uid	Num	4	
	annot	Ref	4	Annot/uid
	perf_volume	Num	4	
	perf_vol_slope	Num	2	
	persistence	Str	1	

Dynamic				
	dynamic_def	Ref	4	DynDef/uid
	chord	Ref	4	Chord/uid
Hairpin				
	hairpin_gd	Ref	4	GraphDef/uid
	group_ref	Ref	4	Group/uid
	perf_vol_slope	Num	2	
Midi				
	event	Ref	4	Event/uid
	command	Str	10	
	command_length	Num	1	
Slur				
	slur_gd	Ref	4	GraphDef/uid
	group_ref	Ref	4	Group/uid
	slur_gr_p1	Num	4	
	slur_gr_p2	Num	4	
Tie				
	tie_gd	Ref	4	GraphDef/uid
	event	Ref	4	Event/uid
Syllable				
	*uid	Num	4	
	sync_par	Ref	4	Sync/uid
	sync_ord	Num	4	
	text_par	Ref	4	Text/uid
	text_ord	Num	4	
	syllable_annot	Ref	4	Annot/uid
Notehead				
	notehead_gd	Ref	4	GraphDef/uid
	note	Ref	4	Note/uid
GrParm				
	*parmname	Str	20	
GDefParm				
	grparm	Ref	20	GrParm/parmname
	graphdef	Ref	4	GraphDef/uid
HomePart				
	staff	Ref	4	Staff/uid
	part	Ref	4	Part/uid
	row	Num	4	

APPENDIX D

Sample Rule Sets for Musical Virtual Attributes

In this appendix, the rule sets used in the examples of section 3.6 are collected together. The example determines the volume attribute of a note. It is presented mostly without comment.

There presentation here is intended to indicate the complexity of what at first might seem a simple inheritance function, as well as the variety of constructs (relational constructs, programming language control structures, and functional apparatus in the style of DAPLEX [Shi81]) needed for these operations.

The following rule set, given a note *n*, determines the volume of the note, volume(*n*), and the rate of change of that volume over time, slope(*n*).

```
/* Find the dynamic which "covers" note n */

range of n is NOTE
range of d is DYNAMIC
retrieve d.uid
   where voice(d) = voice(n)
   and time(d) ≤ time(n) < time(next(d))

dynamic(n) ← d.uid

/* Find the volume of the note n */

volume(n) ← volume(dynamic(n)) +
   slope(n) * (time(n) - time(dynamic(n)));

/* Find the slope of this volume for n: */

slope(note) ← slope(dynamic(n));

if slope_sign = 0
   then slope(d) ← 0.
   done.

if ENDLINEAR ∈ type(d) then
   if (abs_nearby(d))
      slope(d) ← slope(prev(d))
   else
      slope(d) ← 0;
   done

else if (not ENDLINEAR ∈ type(next(d)))
   slope(d) ← (volume(next(d)) − volume(d)) / duration(d)
```

```
else if (abs_nearby(next(d)))
  slope(d) ← (volume(next(next(d))) − volume(d)) / (duration(d) + duration(next(d)))
else /* untagged end with no nearby fixed */
  slope(d) ← (slope_sign(d) * DEFAULT_CRESC) / duration(d);

abs_nearby(vol):
  fixed_nearby(vol) ←
    slope_sign(next(vol)) = 0 &&
    duration(vol) < SMALL_INTERVAL; /* say, 1 second? */

duration(vol):
  duration(vol) ← time(next(vol)) - time(vol);

time(sync):
  if (prev(sync))
    time(sync) ← time(prev(sync)) + duration(prev(sync))
  else
    time(sync) ← 0

voice(note):
  voice(note) ← voice(group(chord(note)));
sync(note):
  sync(note) ← sync(chord(note));
time(note):
  time(note) ← time(sync(note));
time(vol):
  time(vol) ← time(sync(vol));

duration(sync):
  duration(sync) ← minimum(duration(chord)
          where sync(chord) = sync)
```

APPENDIX E

The Ordered Aggregate for Exponential Average

We present here an implementation of the ordered aggregate that maintains the exponential average, as described in section 5.3.1.

E.1. The Averaging Function

An exponential average $\bar{x}$, associated with each element in an ordered set of values,

$$A = \{a_0, a_1, a_2, \cdots, a_n\}.$$

is defined by the following recurrence relation:

$$\bar{x}_0 = sa_0$$
$$\bar{x}_i = sa_i + (1-s)\bar{x}_{i-1}$$

Intuitively, the exponential average $\bar{x}_i$ associated with element a_i is an average of all preceding elements $\{a_0 \cdots a_i\}$. Those elements toward the end of the ordering (that is, close to a_i) are weighted heavily, and those at the beginning are weighted very little.

The rate at which the weight decreases is exponential, and is controlled by the scaling factor, s, $0 < s < 1$. As s increases, the weight of a_i in the average $\bar{x}_i$ is increased.

E.2. Declaring the Aggregate Function

```
define ordered aggregate exp_avg
        (scale = constant float, value = float)
        returns float
        file = "/aggregates/expavg.o"
```

The exponential average aggregate is declared to take two parameters, "scale" and "value" of type float. The scale parameter corresponds to the weighting factor s, and the value parameter corresponds to the elements of the set A.

As an example of the use of this ordered aggregate, we review the structure of the RUNQUEUE relation presented in section 5.3.1:

```
define entity RUNQUEUE (length = integer)
define ordering (RUNQUEUE)
```

This relation stores an ordered set of values, representing the length of a queue sampled at regular time intervals. We associate an attribute representing the exponential average at each point in time, and call it the load:

```
range of q is QUEUE
define inheritance q (load = exp_avg(0.2,q.length))
```

The system will recognize that the scale parameter is taking a constant value, and that the length parameter, when it is used, must be converted from an **integer** to a **float** value.

E.3. User Routines for Exponential Average

We need to provide five routines for an A-tree to efficiently determine the value of $\overline{x}_i$ for every record in a relation. These routines, discussed in section 5.5, will be presented here. The routines are written in the C programming language, though for these examples, the syntax is occasionally modified for the sake of clarity.

E.3.1. InitializeScan

```
typedef float *RESULT;
typedef float *VALUES[];
typedef struct {
        float weighted_sum;
        int             count;
} *STATE;

STATE InitializeScan()
{
        state = alloc_state(sizeof(struct state));
        state -> weighted_sum = 0.0;
        state -> count = 0;
        return(state);
}
```

This routine is called by the system at the beginning of each scan. STATE is a pointer to the state information. The system provides the alloc_state() routine to generate a block of memory that will be recovered automatically when the scan is completed. The state for this aggregate consists of a running weighted sum, and a count of the number of objects currently in the weighted sum.

E.3.2. NextLeaf

```
STATE NextLeaf(state, constant, parameter)
STATE   state;
VALUES constant, parameter;
{
        a ← *parameter[0];
        s ← *constant[0];
        x ← state->weighted_sum;

        x ← (1−s)x+sa;
        state->weighted_sum = x;

        return(state);
}
```

The system calls NextLeaf when scanning data records. The parameter list consists of those nonconstant parameters defined by the **define ordered aggregate** statement. In this example, the only parameter is "value," which is taken from the "length" attribute of the RUNQUEUE records (as specified in the **define inheritance** statement above).

The VALUES structure which contains these parameters is implemented as an array of pointers. In this way, each parameter may be a different type of object. The user routine is responsible for knowing the type of each value, as determined by the **define ordered aggregate** statement.

E.3.3. NextInner

```
STATE NextInner(cumstate,newstate,constant)
STATE cumstate,newstate;
VALUES constant;
{
        x ← cumstate -> weighted_sum;
        i ← cumstate -> count;
        i′ ← newstate -> count;
        x′ ← newstate -> weighted_sum;
        s ← *constant[0];

        x ← x(1−s)^i′ + x′;
        i ← i + i′;

        cumstate->count = i;
        cumstate->weighted_sum = x;

        return(cumstate);
}
```

E.3.4. Result

```
RESULT Result(state)
STATE state;
{
        return(& state -> weighted_sum);
}
```

This routine returns a pointer to the result of the aggregate calculation, namely, the weighted sum.

E.3.5. Compare

```
Compare(r1, r2)
RESULT  r1, r2;
{
        return (*r1 - *r2);
}
```

This routine compares two results, and returns a negative number if the first is less than the second, zero if they are equal, and a positive number if the first result is greater than the second.

In summary, the exponential average aggregate can be implemented quite simply using the interface developed in chapter 5. A total of less than 70 lines of code need be provided by the user for this example.

APPENDIX F

Summary of Proposed Query Language Extensions

The following extensions to QUEL have been proposed in this dissertation to support hierarchical ordering, as defined in chapter 3, and hierarchically ordered relations, as described in chapter 5.

Standard BNF descriptions [Bac59] are used to present the syntax of statements in the extended query language. Words in **boldface** are literal, or key words. Square brackets indicate optional clauses, and curly braces indicate clauses that may be repeated zero or more times.

F.1. Data Definition Language Extensions

We begin with those statements that define entities, attributes, aggregates, and orderings. The data model developed in chapter 3 is implemented by these extensions. A collection of these statements define the schema for a particular database design.

F.1.1. The define entity Statement

The **define entity** statement defines an entity and its associated native attributes.

The syntax for this statement is:

```
define_entity_statement :
        define entity entity_name ( attribute_spec { , attribute_spec } )

attribute_spec :
        attribute_name = type

entity_name :
        relation_name

type :
        entity_name |
        adt_name |
        internal_type
```

The name of the entity defined by the **define entity** statement is "entity_name". The native attributes of this entity are also given in this statement. The type of each attribute is either an built-in "internal_type," such as **i4** for a four byte integer, or **c20** for a 20 character string, or else an abstract

data type "adt_name," defined by the user. The abstract data type must have been previously registered with the system. This is accomplished with a **define adt** statement, as proposed in [Ong82].

If the type of an attribute is specified by an entity_name, then it may be represented internally by a pointer to a record in the entity relation so specified. This is identical to the entity reference specifications of GEM [Zan83].

F.1.2. The define ordering Statement

The **define ordering** statement specifies that the entities in a given relation are to be considered as an ordered set, or a hierarchically ordered set. Its syntax is:

define_ordering_statement :
define ordering [order_name] (child_entity { , child_entity })
[**under** parent_entity]

order_name :
relation_name

child_entity :
entity_name

parent_entity :
entity_name

This statement defines an ordering, named "order_name" over entities whose types are given as the "child_entity" parameters. If there are multiple children, then the ordering is inhomogeneous (entities from different relations participate in the ordering).

If the **under** clause is included, then the ordering is hierarchical. Every instance of a child entity is associated with an instance type "parent_entity." The children are partitioned by parent, and ordered within their partition.

The system will generate an ordered relation for each **define ordering** statement. The "order_name" field may be omitted if,

(1) the ordering contains only a single "child_entity" entry, and

(2) that child entity is not currently an ordered relation.

If these conditions hold, and the order_name is omitted, then the base relation indicated by the "child_entity" is defined to be an ordered relation. The relation name associated with the child entity

may then be used as an "order_name" in future definitions (i.e. the system assigns the child entity name as the "order_name" for an unnamed ordering).

If an order_name is specified, then a new relation is defined as a secondary index over the child entities. This index is a relation whose attribute is a pointer (i.e. tuple identifier, or *TID*) to a child entity.

If the parent_name is specified in an **under** clause, then a pointer to a parent entity is included as an attribute in the resultant ordered relation.

F.1.3. The define aggregate Statement

User defined aggregates are registered with the system using the **define aggregate** statement. Its syntax is:

```
define_aggregate_statement :
        define [ ordered ] aggregate aggregate_name
                                [ ( parameter_spec { , parameter_spec } ) ]
                                returns return_type
                                [ ascending | descending ]
                                file = file_name

parameter_spec :
        formal_parameter_name = parameter_type

return_type :
        entity_name |
        adt_type |
        internal_type |
        typeof ( formal_parameter_name )

parameter_type :
        entity_name |
        adt_type |
        internal_type |
        constant
```

If the **ordered** keyword is omitted from the define aggregate statement, then this command is implemented as a user-defined aggregate as specified in [Han84]. The "aggregate_name" is the name the user wishes to associate with the aggregate function whose implementation is stored in the file "file_name." The remaining clauses characterize the aggregate function implemented in this file.

A call to an aggregate function defines a set of records over which an aggregate value is to be calculated. Some of the parameters to the aggregate function are taken from attribute values in each

record, and some are constant over the set, independent of the records over which the aggregate is calculated. Each such parameter is given a "formal_parameter_name" and a "parameter_type." If the type is **constant**, then that parameter is fixed over the aggregate calculation. Otherwise, the type is defined to be the type of the attribute value taken from each record over which the aggregate will be calculated.

Unlike aggregate functions which calculate a single value based on a set of records, an ordered aggregate function determines a distinct value for each record in its set. The value for a given record depends on the attributes of that record and all records previous to it in the ordering.

If the result of an ordered aggregate function is guaranteed to increase monotonically over its ordered set, then it may be declared as **ascending**. If it decreases, it may be declared as **descending**. These clauses allow for more efficient processing by the system when the ordered aggregate functions meet these special criteria.

F.1.4. The define inheritance Statement

The define inheritance statement associates inherited attributes with an entity. These attributes are added to the set of attributes already defined for the given entity. The syntax for inheritance definition is:

define_inheritance_statement :
define inheritance range_variable (target_list)
where qualification

This syntax is similar to the syntax for the **replace** statement, but its effect is to define additional attribute values, rather than to replace existing attribute values. The expressions which appear in the "target_list" and "qualification" are not evaluated at this time, but at the time the inherited attribute is accessed. In this respect, they are similar to views [Sto75], except that they are defined at the attribute level, rather than at the relation level.

F.2. Data Manipulation Language Extensions

Given a database schema, the various manipulations of its data are accomplished using the data manipulation language of QUEL, which permits retrieval, update, and insertion into the database. We also include the statements which allow the user to define the storage type of database relations in this

section.

F.2.1. The modify Statement

A new storage type has been introduced, the A-tree, and so an additional form of the **modify** command is provided to convert an ordered relation to this storage type:

modify order_name **to A-tree**

An "order_name" indicates an ordered relation. It was defined using the **define ordering** command. As was mentioned, the name of the child entity in an unnamed ordering may be used as an order_name in this context (this is the case where a base relation itself is an ordered relation).

F.2.2. The reorder Statement

The position of records in an ordered relation may be determined at the time each record is inserted (using the **append before** or **append after** constructs). Alternatively, the records may be inserted in arbitrary order, and the **reorder statement** may then be used to establish an order based on the sort order of an attribute within the relation.

The syntax is:

reorder order_name **on** attribute_name

The effect is to order all the entities participating in the given ordering according to the ordering induced by sorting the entities on attribute_name.

F.2.3. User-Defined Aggregate Expressions

Expressions in queries, appearing either in the target list or the qualification, may contain references to user-defined aggregate functions. Ordered aggregates are invoked as terms within an expression using the following syntax:

ordered_aggregate_term :
ordered_aggregate_name (parameter { , parameter }
[**in** order_name] [**where** qualification])

This syntax is similar to those for existing aggregates in QUEL, except that:

(1) multiple parameters may be specified. If the corresponding formal parameter in the **define aggregate** statement is of type **constant**, then this actual parameter must be a constant. Otherwise, it must be an attribute (of a relation) whose type is consistent with the type of the corresponding formal parameter.

(2) The **by** clause, which is used to partition regular aggregate functions, is not applicable for ordered aggregates.

(3) The ordering for the ordered aggregate function may be explicitly given in an **in** clause. If it is not given, the parameters must all be taken from a single ordered relation which implicitly defines the ordering for the aggregate.

F.2.4. Expressions of Type "Entity"

As in GEM, we permit expressions to operate on entities themselves, and expressions to evaluate to an entity value. An expression of type entity may be

(1) an attribute value whose type was defined as "entity_name,"

(2) a range variable, whose value at any point in the scan of an entity relation is the current record of that range variable. The type of such an entity is determined syntactically by the **range** statement where the range variable is declared.

(3) An operation that evaluates to an entity value, for example, the **first** and **last** aggregate functions.

All three of these objects are syntactically equivalent. We therefore allow expressions such as "*(a.b).c*" where a is a relation name, b is an attribute of a whose type is entity x, and c is an attribute of x.

F.2.5. Comparison of Entities

The qualification of a query consists of a series of boolean terms of the form "aoperatorb," where operator is one of the comparison operators ($<$, $\leq$, $=$, $>$, or $\geq$). These terms are combined using the boolean operators **and**, **or**, and **not**. We extend the set of boolean operators with those that compare entities, rather than values. Because the ordering of entities is not intrinsic to the entities (as the ordering of, say, integers is intrinsic), but rather is determined by a user-defined ordering, the

comparison operators for entities take three parameters: the two entities to be compared, and the ordering by which they are to be compared. The syntax for these comparisons is as follows:

boolean_term :
expression entity_comparator expression [**in** order_name]

entity_comparator :
is | **before** | **after** | **under**

In order to support entity comparison, we extend the query language to include expressions that result in objects of type "entity." When we compare two such expressions,

- **is** evaluates to "true" if the expressions refer to the same entity,
- **before** evaluates to "true" if the two entities are comparable, and the first entity is before the second one in the ordering specified by the **in** clause,
- **after** evaluates to "true" if the two entities are comparable, and the first entity is after the second one in the ordering specified by the **in** clause, and
- **under** evaluates to "true" if the two entities are comparable, and the first entity is under the second one in the hierarchical ordering specified by the **in** clause.

Two entities are comparable if they participate in the same ordering. For **before** and **after**, the entities must both participate as children in the ordering. For **under**, the first entity must be a child in the ordering, and the second entity must be a parent. Thus, in this case, the ordering must be hierarchical. Whenever two entities are not comparable, the entity_comparator operations evaluate to "false."

F.2.6. The append Statement

The append statement is modified to allow for insertion of records at a particular location within an ordering.

append_statement :
append relation_name
[location]
[(target_list)]
where qualification

location :
[**before** entity]
[**after** entity]
[**under** entity] **in** order_name

In order for a "location" to be specified for an insertion, the "relation_name" must refer to an ordered relation. The location then pinpoints the record after which, before which, or under which (in the case of hierarchical orderings) the insertion should be performed. If more than one of these locator clauses is specified, the location may be over-constrained. In other words, there may not exist an insertion point such that the neighboring records each satisfy the location constraints. In this case, the append statement is non-functional.

If multiple entities in the database satisfy the qualification, then this set of entities must form an ordered relation, which will be properly inserted *en-masse* at the given location. The ordering of these entities will be preserved by the system (the batch insertion algorithms presented in [CDR86] may be used for this purpose).

F.2.7. The replace Statement

The replace statement is extended in a similar manner:

replace entity
[location]
[(target_list)]
where qualification

If the entity is a member of an ordered relation, then the location statement may be specified to set a new location for the entity within the ordering. The target list is now optional. It makes sense to change the location of an entity without modifying any of its attribute values. The qualification determines the set on which the replacement will occur. If multiple entities satisfy the qualification, and location is specified, then they themselves must form an ordered relation, which will be correctly inserted into the ordering at the new location.

References

[Ado85] Adobe Systems, *PostScript Language Reference Manual*, Addison-Wesley, Reading, MA, 1985.

[Alp80] Alphonce, B., "Music Analysis by Computer", *Computer Music Journal 4*, 2 (Summer 1980), 26-35.

[And81] Anderson, T. L., *The Database Semantics of Time*, Ph.D. Dissertation, University of Washington, 1981.

[AnK86a] Anderson, D. and Kuivila, R., "A Model of Real-Time Computation for Computer Music", *Proceedings of the International Computer Music Conference*, The Hague, Netherlands, 1986, 35-42.

[AnK86b] Anderson, D. and Kuivila, R., *FORMULA on the Atari ST*, (no listed publisher), October 1986.

[AJM84] Aragon, C., Johnson, D., MeGeoch, L. and Schevon, C., "Optimization By Simulated Annealing: An Experimental Evaluation", Technical Report Draft, September 1984.

[ACJ83] Ariav, G., Clifford, J. and Jarke, M., "Panel on Time and Databases", *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*, Ann Arbor, MI, May 1983, 243-245.

[Ash83] Ashley, R., "Production Systems: Three Applications in Music", *Proceedings of the International Computer Music Conference*, Rochester, NY, 1983, 160-172.

[Ash85] Ashley, R. D., "KSM: An Essay in Knowledge Representation in Music", *Proceedings of the International Computer Music Conference*, Burnaby, British Columbia, 1985, 383-390.

[Bac47] Bach-Gesellschaft, *Johann Sebastian Bach's Werke*, Breitkopf Haertel, Leipzig, 1947.

[Bac59] Backus, J., "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference", *Proceedings of the International Conference on*

Information Processing, 1959, 125-132.

[BaP85] Barbic, F. and Pernici, B., "Time Modeling in Office Information Systems", *Proceedings of the ACM-SIGMOD International Conference on the Management of Data 14*, 4 (December 1985), 51-62.

[BaM72] Bayer, R. and McCreight, E., "Organization and maintenance of large ordered indexes", *Acta Informatica 1* (1972), 173-189.

[Bee60] Beethoven, L., *Symphonies No. 5, 6, 7*, Edwin Kalmus, 1960.

[BoW77] Bobrow, D. and Winograd, T., "An Overview of KRL: Knowledge Representation Language", *Cognitive Science 1*, 1 (1977), 2-46.

[BoS81] Bobrow, D. and Stefik, M., "The Loops Manual", Technical Report KB-VLSI-81-13, Xerox Palo Alto Research Center, Palo Alto, CA, 1981.

[BAD82] Bolour, A., Anderson, T., Dekeyser, L. and Wong, H., "The Role of Time in Information Processing: A Survey", *ACM SIGMOD Record 12*, 3 (1982), 27-50.

[BDR85] Braegger, R., Dudler, A., Rebsamen, J. and Zehnder, C., "Gambit: An Interactive Database Design Tool for Data Structures, Integrity Constraints, and Transactions", *IEEE Transactions on Software Engineering SE-11*, 7 (July 1985), 574-583.

[Bru72] Bruce, B. C., "A Model for Temporal References and Its Application in a Question Answering Program", *Artificial Intelligence 3* (1972), 1-25.

[BRB78] Buxton, W., Reeves, W., Baeker, R. and Mezei, L., "The Use of Hierarchy and Instance in a Data Structure for Computer Music", *Computer Music Journal 2*, 4 (Winter 1978), 10-20.

[BSR79] Buxton, W., Sniderman, R., Reeves, W., Patel, S. and Baeker, R., "The Evolution of the SSSP Score Editing Tools", *Computer Music Journal 3*, 4 (1979), 14-26.

[BPR81] Buxton, W., Patel, S., Reeves, W. and Baecker, R., "Scope in Interactive Score Editors", *Computer Music Journal 5*, 3 (Fall 1981), 50-56.

[Byr84] Byrd, D., *Music Notation By Computer*, Ph.D. Dissertation, Department of Computer Science, Indiana University, 1984.

[CDR86] Carey, M., DeWitt, D., Richardson, J. and Shekita, E., "Object and File Management in the EXODUS Extensible Database System", *Proceedings of the International Conference on Very Large Data Bases*, Kyoto, August 1986.

[Car62] Carter, E., *Double Concerto for Harpsichord and Piano with Two Chamber Orchestras*, Associated Music Publishers, New York, 1962.

[CMR82] Chafe, C., Mont-Reynaud, B. and Rush, L., "Toward an Intelligent Editor of Digital Audio: Recognition of Musical Constructs", *Computer Music Journal 6*, 1 (Spring 1982), 30-41.

[Che76] Chen, P., "The Entity-Relationship Model: Toward a Unified View of Data", *ACM Transactions on Database Systems 1*, 1 (March 1976), 9-36.

[ClW83] Clifford, J. and Warren, D., "Formal Semantics for Time in Databases", *ACM Transactions on Database Systems 8*, 2 (June 1983), 214-254.

[Cod70] Codd, E., "A Relational Model of Data for Large Shared Data Banks", *Communications of the ACM 13*, 6 (June 1970), 377-387.

[Cod79] Codd, E. F., "Extending the Database Relational Model to Capture More Meaning", *ACM Transactions on Database Systems 4*, 4 (December 1979), 397-434.

[Com79] Comer, D., "The Ubiquitous B-Tree", *ACM Computing Surveys 11*, 2 (June 1979), 121-137.

[CoM84] Copeland, G. and Maier, D., "Making Smalltalk a Data Base System", *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*, Boston, MA, June 1984, 316-325.

[Dan86] Dannenberg, R., "A Structure for Representing, Displaying and Editing Music", *Proceedings of the International Computer Music Conference*, The Hague, Netherlands, 1986, 153-160.

[DeK85] Decker, S. L. and Kendall, G. S., "A Unified Approach to the Editing of Time-Ordered Events", *Proceedings of the International Computer Music Conference*, Burnaby, British Columbia, 1985, 69-78.

[DeF84] Deering, M. and Faletti, J., "Database Support for Storage of AI Reasoning Knowledge", *Proceedings of the First International Workshop on Expert Data Base Systems*, Kiawah, SC, October 1984.

[Don63] Donato, A., *Preparing Music Manuscript*, Prentice-Hall, Englewood Cliffs, NJ, 1963.

[Ebc84] Ebcioglu, K., "An Expert System for Schenkerian Synthesis of Chorales in the Style of J. S. Bach", *Proceedings of the International Computer Music Conference*, Paris, 1984, 233-242.

[Eri77] Erickson, R., *DARMS: A Reference Manual*, (no listed publisher), 1977.

[ErW83] Erickson, R. and Wolff, A., "The DARMS Project: Implementation of an Artificial Language for the Representation of Music", *Trends in Linguistics 19* (1983).

[FiM82] Fiduccia, C. and Mattheyses, R., "A Linear-Time Heuristic for Improving Network Partitions", *Proceedings of the 19th Design Automation Conference*, 1982, 175-181.

[FiC73] Findler, N. V. and Chen, D., "On the Problems of Time, Retrieval of Temporal Relations, Causality, and Coexistence", *International Journal of Computer and Information Sciences 2*, 3 (1973), 161-185.

[Fog82] Fogg, D., "Implementation of Domain Abstraction in the Relational Database System INGRES", Masters Report, Department of Electrical Engineering and Computer Science, University of California Berkeley, Berkeley, CA, November 1982.

[Fox79] Fox, M. S., "On Inheritance in Knowledge Representation", *Proceedings of the Sixth International Joint Conference on Artifical Intelligence*, Tokyo, 1979, 282-284.

[FWA84] Fox, M. S., Wright, J. M. and Adam, D., "Experiences with SRL: An Analysis of a Frame-based Knowledge Representation", Intelligent Systems Laboratory Technical Report, Robotics Institute, Carnegie-Mellon University, Pittsburgy, PA, June 1984.

[Fry84] Fry, C., "Flavors Band: A Language for Specifying Musical Style", *Computer Music Journal 8*, 4 (Winter 1984), 20-34.

[GoR83] Goldberg, A. and Robson, D., *Smalltalk-80: The language and its Implementation*, Addison-Wesley, Reading, MA, 1983.

[Gom77] Gomberg, D. A., "A Computer-Oriented System for Music Printing", *Computers and the Humanities 11* (1977), 63-80. This article is based on the author's D.Sc. dissertation from Washington University (1975) of the same title.

[Gro84] Gross, D., "Computer Applications to Music Theory: A Retrospective", *Computer Music Journal 8*, 4 (Winter 1984), 35-42.

[Had75] Hadlock, F., "Finding a Maximum Cut of a Planar Graph in Polynomial Time", *SIAM Journal of Computing 4*, 3 (September 1975), 221-225.

[Han84] Hanson, E., "User-Defined Aggregates in the Relational Database System INGRES", Masters Report, Computer Science Division, University of California Berkeley, Berkeley, CA, December 1984.

[HSW75] Held, G., Stonebraker, M. and Wong, E., "INGRES – A Relational Database System", *Proceedings of the National Computer Conference*, Anaheim, CA, May 1975, 409-416.

[HeS86] Hewlett, W. and Selfridge-Field, E., *Directory of Computer Assisted Research in Musicology*, Center for Computer Assisted Research in the Humanities, Menlo Park, CA, June 1986.

[Hil70] Hiller, L., "Music Composed with Computers: A Historical Survey", in *The Computer and Music*, H. Lincoln (editor), Cornell University Press, Ithaca, NY, 1970, 42-96.

[Hug86] Huggins, C., *Symphony: A Font for Music*, Adobe Systems, 1986.

[IBM66] IBM, "OS ISAM Logic", Technical Report GY28-6618, IBM Corporation, White Plains, NY, June 1966.

[ISW84] Ioannidis, Y., Shinkle, L. and Wong, E., "Enhancing INGRES with Deductive Power: A Position Paper", *Proceedings of the First International Workshop on Expert Data Base Systems*, Kiawah, SC, October 1984.

[IoW85] Ioannidis, Y. and Wong, E., "An Algebraic Approach to Recursive Inference", Electronic Research Laboratory Memorandum M85/93, University of California Berkeley, Berkeley, CA, December 1985.

[Jaf85] Jaffe, D., "Ensemble Timing in Computer Music", *Computer Music Journal 9*, 4 (Winter 1985).

[Jun83] Junglieb, S., "MIDI Hardware Fundamentals", *Polyphony 8*, 4 (1983), 34-38.

[Kar72] Karkoschka, E., *Notation in New Music*, Praeger Publishers, New York, 1972. trans. R. Koenig.

[KaL82] Katz, R. and Lehman, T., "Storage Structures for Versions and Alternatives", Computer Sciences Technical Report #479, University of Wisconsin, Madison, July 1982.

[KeL70] Kernighan, B. and Lin, S., "An Efficient Heuristic Procedure for Partitioning Graphs", *Bell System Technical Journal 49*, 2 (February 1970), 291-307.

[Klo83] Klopprogge, M. R., "Term: An Approach to Include the Time Dimension in the Entity-Relationship Model", in *Entity-Relationship Approach to Information Modeling and Analysis*, P. Chen (editor), Elsevier Science Publishers, Amsterdam, 1983, 473-508.

[Knu86] Knuth, D., *The METAFONT Book*, Addison-Wesley, Reading, MA, 1986.

[Kra79] Krasner, M. A., *Digital Encoding of Speech and Audio Signals Based on the Perceptual Requirements of the Auditory System*, Ph.D. Dissertation, Lincoln Laboratory, Massachusetts Institute of Technology, Lexington, MA, June 1979.

[KuM77] Kundu, S. and Misra, J., "A Linear Tree Partitioning Algorithm", *SIAM Journal of Computing 6*, 1 (March 1977), 151-154.

[LeG78] Lee, R. M. and Gerritsen, R., "Extended Semantics for Generalization Hierarchies", *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*, Austin, TX, May 1978, 18-25.

[Leo81] Leonard, H., *Broadway! The Best From Broadway's Top Shows*, Hal Leonard Publishing Corporation, 1981.

[Lin77] Lincoln, H. B., "Encoding, Decoding and Storing Melodies for a Data Base of Renaissance Polyphony: A Progress Report", *Proceedings of the Third International Conference on Very Large Data Bases*, Tokyo, October 1977, 277-282.

[LoA85] Loy, G. and Abbott, C., "Programming Languages for Computer Music Synthesis, Performance, and Composition", *ACM Computing Surveys 17*, 2 (June 1985), 235-265.

[Luk74] Lukes, J., "Efficient Algorithm for the Partioning of Trees", *IBM Journal of Research and Development 18*, 3 (May 1974), 217.

[Luk75] Lukes, J., "Combinatorial Solution to the Partitioning of General Graphs", *IBM Journal of Research and Development 19*, 2 (March 1975), 170.

[LDE84] Lum, V., Dadam, P. and Erbe, R., "Designing DBMS Support for the Temporal Dimension", *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*, Boston, MA, June 1984, 115-130.

[Lyn82] Lynn, N., "Implementation of Ordered Relations in a Data Base System", Masters Report, Department of Electrical Engineering and Computer Science, University of California Berkeley, Berkeley, CA, September 1982.

[Mac78] MacGregor, R., *On Partitioning a Graph: A Theoretical and Empirical Study*, Ph.D. Dissertation, Computer Science Division, University of California Berkeley, Berkeley, CA, June 1978.

[MRW86] Maier, D., Rozenshtein, D. and Warren, D., "Window Functions", in *Advances in Computing Research*, P. Kanellakis (editor), JAI Press, London, 1986, 213-246.

[Mat69] Mathews, M., *The Technology of Computer Music*, Massachusetts Institute of Technology Press, Cambridge, MA, 1969.

[MaM70] Mathews, M. and Moore, F., "GROOVE – A Program to compose, store and edit functions of time", *Communications of the ACM 13*, 12 (December 1970), 715-721.

[MaO83] Maxwell, J. T. and Ornstein, S. M., "Mockingbird: A Composer's Amanuensis", Technical Report CSL-83-2, Xerox Palo Alto Research Center, Palo Alto, CA, January 1983.

[McL86a] McLean, B., "The DARMS Cube: The Design of a Data Structure for Score Processing Applications", *Symposium on Computers and Music Research*, Oxford, July 1986.

[McL86b] McLean, B., *A Database System for Score-Processing Applications in Musical Computing*, Ph.D. Dissertation, State University of New York, Binghamton, 1986. In preparation.

[Men84] Mendelzon, A., "Database states and their tableaux", *ACM Transactions on Database Systems 9*, 2 (1984), 264-282.

[MoP64] Moder, J. J. and Philips, C. R., *Project Management with CPM and PERT*, Reinhold, New York, 1964.

[Moo85] Moore, F. R., "The Cmusic Sound Synthesis Program", Computer Audio Research Laboratory Technical Report, University of California, San Diego, La Jolla, CA, 1985.

[Ong82] Ong, J., "The Design and Implementation of Abstract Data Types in the Relational Database System INGRES", Masters Report, Department of Electrical Engineering and Computer Science, University of California Berkeley, Berkeley, CA, September 1982.

[Ong83] Ong, J., "Implementation of Data Abstraction in the Relational Database System INGRES", *ACM-SIGMOD International Conference on the Management of Data*, 1983.

[OpS75] Oppenheim, A. and Schafer, R., *Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1975.

[Ove82] Overmyer, R., "Implementation of a Time Expert in a Database System", *ACM SIGMOD Record 12*, 3 (1982), 51-60.

[Pap79] Papadimitriou, C., "The Serializability of Concurrent Database Operations", *JACM 26*, 4 (1979), 631-653.

[Pri67] Prior, A., in *Past, Present, Future*, Oxford University Press, 1967.

[Pru84a] Prusinkiewicz, P., "INTERSCORE – An interactive score editor for microcomputers", *Proceedings of the Fourth Symposium on Small Computers in the Arts*, Philadelphia, PA, 1984, 58-64.

[Pru84b] Prusinkiewicz, P., "Time Management in Interactive Score Editing", *Proceedings of the International Computer Music Conference*, Paris, 1984, 275-280.

[Rea69] Read, G., *Music Notation*, Allyn and Bacon, Boston, 1969.

[Rel84] Relational Technology Incorporated, *INGRES Reference Manual, Version 2.1*, Relational Technology Incorporated, Alameda, CA, July 1984.

[ReU71] Rescher, N. and Urquhart, A., in *Temporal Logic*, Springer Verlag, New York, 1971.

[Roa79] Roads, C., "Grammars as Representations for Music", *Computer Music Journal 3*, 1 (March 1979), 48-56.

[Roa85] Roads, C., "Research in Music and Artificial Intelligence", *ACM Computing Surveys 17*, 2 (June 1985), 163-190.

[RoC84] Rodet, X. and Cointe, P., "FORMES: Composition and Scheduling of Processes", *Computer Music Journal 8*, 3 (Fall 1984), 32-50.

[Rol85] Roland, *MPU-401 Technical Reference Manual*, Roland DG Corporation, 1985.

[RSS84] Romeo, F., Sechen, C. and Sangiovanni-Vincentelli, A., "Simulated Annealing Research at Berkeley", *Proceedings of the International Conference on Computer Design*, Port Chester, NY, 1984.

[Ros70] Ross, T., *The Art of Music Engraving and Processing*, Hansen Books, Miami, 1970.

[RoL85] Roussopoulos, N. and Leifker, D., "Direct Spatial Search on Pictorial Databases Using Packed R-trees", *Proceedings of the ACM-SIGMOD International Conference on the Management of Data 14*, 4 (December 1985), 17-31.

[Rub85] Rubenstein, W. B., "Indices for Time-Ordered Data", Masters Thesis, Computer Science Division, University of California Berkeley, Berkeley, CA, May 1985.

[Sag83] Sagiv, Y., "A characterization of globally consistent databases and their correct access paths", *ACM Transactions on Database Systems 8*, 2 (1983), 266-286.

[Sch50] Schmieder, W., *Thematische-systematisches Verzeichnis der musikalischen Werker von Johann Sebastian Bach*, Breitkopf Haertel, Leipzig, 1950.

[Sch83] Schottstaedt, B., "Pla: A Composer's Idea of Language", *Computer Music Journal 7*, 1 (1983), 11-20.

[Sch77] Schueler, B., "Update Reconsidered", in *Architecture and Models in Data Base Management Systems*, Nijsson (editor), North-Holland Publishing Company, Amsterdam, 1977, 129-161?.

[Shi81] Shipman, D., "The Functional Data Model and the Data Language DAPLEX", *ACM Transactions on Database Systems 6*, 1 (March 1981), 140-173.

[STZ84] Shmueli, O., Tsur, S. and Zfira, H., "Rule Support in Prolog", *Proceedings of the First International Workshop on Expert Data Base Systems*, Kiawah, SC, October 1984, 547-565.

[Sho82] Shoshani, A., "Statistical Databases: Characteristics, Problems, and Some Solutions", *Proceedings of the International Conference on Very Large Data Bases*, Mexico City , 1982, 208-222.

[SOW84] Shoshani, A., Olken, F. and Wong, H., "Characteristics of Scientific Databases", *Proceedings of the International Conference on Very Large Data Bases*, 1984, 147-160.

[ShK86] Shoshani, A. and Kawagoe, K., "Temporal Data Management", Technical Report LBL-21143, Lawrence Berkeley Laboratory, Berkeley, CA, February 1986.

[Smi72] Smith, L., "SCORE – A Musician's Approach to Computer Music", *Journal of the Audio Engineering Society 20*, 1 (January 1972), 7-14.

[Smi73] Smith, L., "Editing and Printing Music by Computer", *Journal of Music Theory 17*, 2 (1973), 292-308.

[Sno84] Snodgrass, R., "The Temporal Query Language TQuel", *Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Data Base Systems*, Waterloo, Ontario, April 1984, 204-212.

[SnA85] Snodgrass, R. and Ahn, I., "A Taxonomy of Time in Databases", *Proceedings of the ACM-SIGMOD International Conference on the Management of Data 14*, 4 (December 1985), 236-246.

[SLR76] Stearns, R., Lewis, P. and Rosenkrantz, D., "Concurrency Control for Database Systems", *Proceedings of the IEEE Symposium on Foundations of Computer Science*,

1976, 19-32.

[StB86] Stefik, M. and Bobrow, D., "Object-Oriented Programming: Themes and Variations", *AI Magazine 6*, 4 (Winter 1986), 40-62.

[Sto75] Stonebraker, M., "Implementation of Integrity Constraints and Views by Query Modification", *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*, San Jose, CA, June 1975. Also available as Electronic Research Laboratory Memorandum M514, March 1975.

[StR80] Stonebraker, M. and Rowe, L., "Database Portals: A New Application Program Interface", Electronic Research Laboratory Memorandum M82/80, University of California Berkeley, Berkeley, CA, November 1980.

[SSK82] Stonebraker, M., Stettner, H., Kalash, J., Guttman, A. and Lynn, N., "Document Processing in a Relational Data Base System", Electronic Research Laboratory Memorandum M82/32, University of California Berkeley, Berkeley, CA, May 1982.

[SRG83] Stonebraker, M., Rubenstein, W. B. and Guttman, A., "Application of Abstract Data Types and Abstract Indices to CAD Databases", *Proceedings of the Engineering Design Applications of ACM-IEEE Data Base Week*, San Jose, CA, May 1983. Also available as Electronic Research Laboratory Memorandum M83/3 from University of California Berkeley.

[SAH85] Stonebraker, M., Anton, J. and Hanson, E., "Extending a Data Base System with Procedures", Electronic Research Laboratory Memorandum M85/59, University of California Berkeley, Berkeley, CA, 1985.

[StR85] Stonebraker, M. and Rowe, L., "The Design of POSTGRES", Electronic Research Laboratory Memorandum M85/95, November 1985.

[SSH86] Stonebraker, M., Sellis, T. and Hanson, E., "An Analysis of Rule Indexing Implementations in Database Systems", *Proceedings of the First International Conference on Expert Data Base Systems*, Kiawah, SC, April 1986.

[Str68] Strauss, J., *Die Fledermaus*, Edition Eulenburg GmbH, Zurich, 1968.

[Tho85] Thomas, M. T., "Vivace: A Rule Based AI System for Composition", *Proceedings of the International Computer Music Conference*, Burnaby, British Columbia, 1985, 267-275.

[TsZ84] Tsur, S. and Zaniolo, C., "An Implementation of GEM – supporting a semantic data model on a relational back-end", *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*, Boston, MA, June 1984, 286-295.

[Wen77] Wenker, J., *An Analytical Study of Anglo-Canadian Folksong*, Ph.D. Dissertation, Indiana University, 1977.

[Wil85] Wilson, T. A., "Data Reduction of Musical Signals", *Proceedings of the International Computer Music Conference*, Burnaby, British Columbia, 1985, 25-32.

[Win75] Winograd, T., "Frame Representation and the Declarative/Procedural Controversy", in *Representation and Understanding*, D. Bobrow and A. Collins (editor), Academic Press, New York, 1975, 185-210.

[Wol77] Wolff, A. B., "Problems of Representation in Musical Computing", *Computers and the Humanities 11* (1977), 3-12.

[WoY76] Wong, E. and Youssefi, K., "Decomposition - A Strategy for Query Processing", *ACM Transactions on Database Systems 1*, 3 (September 1976), 223-241.

[Yao78] Yao, A., "On Random 2-3 Trees", *Acta Informatica 9*, 2 (1978), 159-170.

[Zan83] Zaniolo, C., "The Database Language GEM", *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*, Ann Arbor, MI, May 1983.